W3School Node.js 教程

wizardforcel

Published with GitBook



目錄

介紹 0
Node.js 简介 1
Node.js 安装配置 2
Node.js 创建第一个应用 3
NPM 使用介绍 4
Node.js REPL(交互式解释器) 5
Node.js 回调函数 6
Node.js 事件循环 7
Node.js EventEmitter 8
Node.js Buffer(缓冲区) 9
Node.js Stream(流) 10
Node.js模块系统 11
Node.js 函数 12
Node.js 路由 13
Node.js 全局对象 14
Node.js 常用工具 util 15
Node.js 文件系统 16
Node.js GET/POST请求 17
Node.js 工具模块 18
Node.js OS 模块 19
Node.js Path 模块 20
Node.js Net 模块 21
Node.js DNS 模块 22
Node.js Domain 模块 23
Node.js Web 模块 24
Node.js Express 框架 25
Node.js RESTful API 26
Node.js 多进程 27
Node.js JXcore 打包 28
免责声明 29

W3School Node.js教程

来源：Node.js教程

整理：飞龙

Node.js 简介



简单的说 Node.js 就是运行在服务端的 JavaScript。

Node.js 是一个基于Chrome JavaScript 运行时建立的一个平台。

Node.js是一个事件驱动I/O服务端JavaScript环境，基于Google的V8引擎，V8引擎执行Javascript的速度非常快，性能非常好。

谁适合阅读本教程？

如果你是一个前端程序员，你不懂的像PHP、Python或Java等动态编程语言，然后你想创建自己的服务，那么Node.js是一个非常好的选择。

Node.js 是运行在服务端的 JavaScript，如果你熟悉Javascript，那么你将会很容易的学会Node.js。

当然，如果你是后端程序员，想部署一些高性能的服务，那么学习Node.js也是一个非常好的选择。

学习本教程前你需要了解

在继续本教程之前，你应该了解一些基本的计算机编程术语。如果你学习过Javascript,PHP，Java等编程语言，将有助于你更快的了解Node.js编程。

第一个Node.js程序：Hello World！

脚本模式

以下是我们的第一个Node.js程序：

```
console.log("Hello World");
```

保存该文件，文件名为 helloworld.js， 并通过 node命令来执行：

```
node helloworld.js
```

程序执行后，正常的话，就会在终端输出 Hello World。

交互模式

打开终端，键入node进入命令交互模式，可以输入一条代码语句后立即执行并显示结果，例如：

```
$ node
> console.log('Hello World!');
Hello World!
```

Node.js 安装配置

本章节我们将向大家介绍在window和Linux上安装Node.js的方法。

本安装教程以Node.js v0.10.26 版本为例。

Node.js安装包及源码下载地址为：http://www.nodejs.org/download/。



Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.exe)	32-bit	64-bit
Mac OS X Installer (.pkg)	Universal	
Mac OS X Binaries (.tar.gz)	32-bit	64-bit
Linux Binaries (.tar.gz)	32-bit	64-bit
SunOS Binaries (.tar.gz)	32-bit	64-bit
Source Code	node-v0.10.26.tar.gz	

Note: Python 2.6 or 2.7 is required to build from source tarballs.

根据不同平台系统选择你需要的Node.js安装包。

注意：Linux上安装Node.js需要安装Python 2.6 或 2.7， 不建议安装Python 3.0以上版本。

Windowv 上安装Node.js

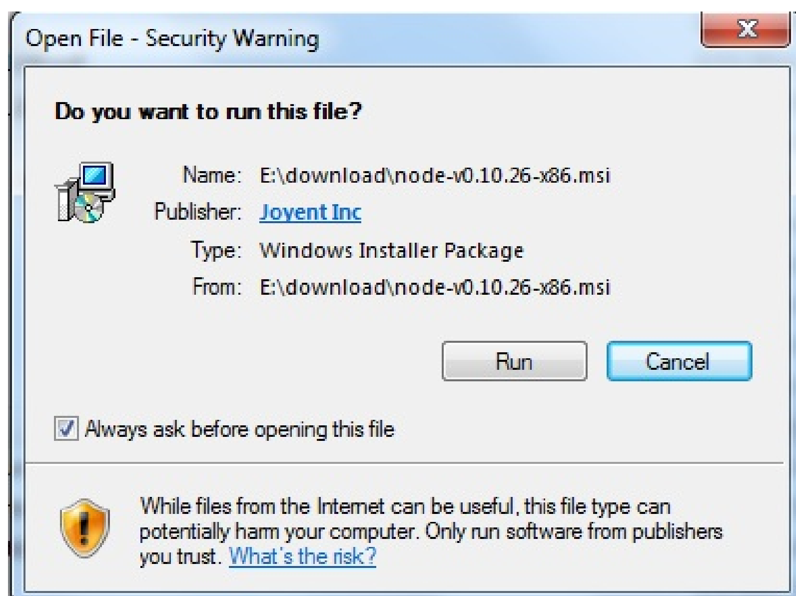
Windows 安装包(.msi) :

32 位安装包下载地址：http://nodejs.org/dist/v0.10.26/node-v0.10.26-x86.msi

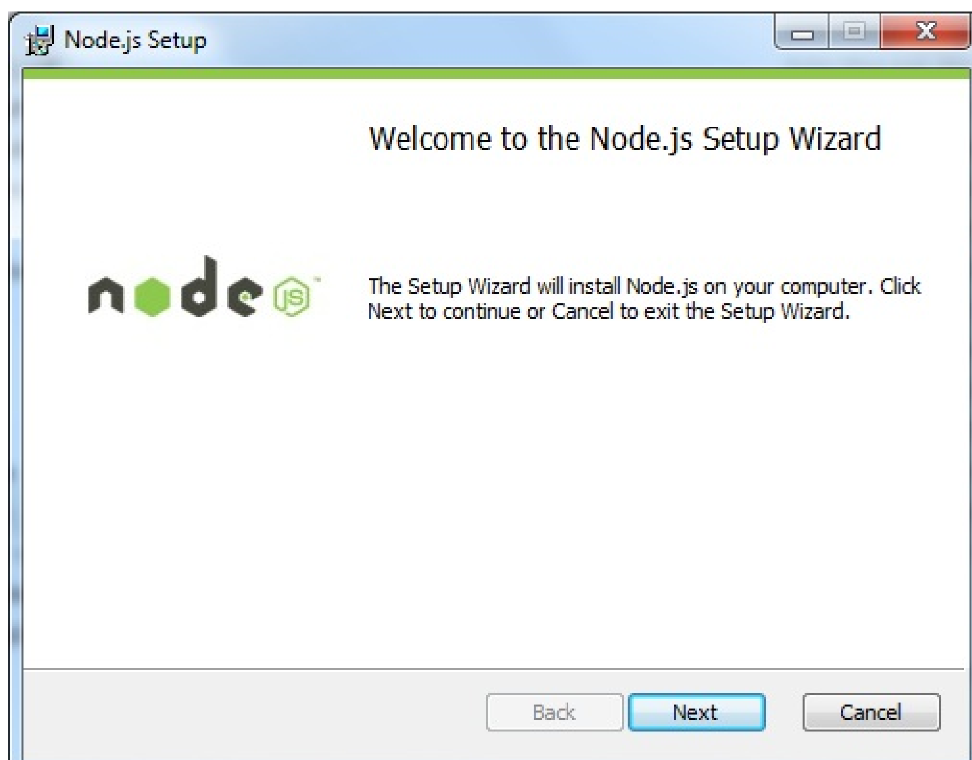
64 位安装包下载地址：http://nodejs.org/dist/v0.10.26/x64/node-v0.10.26-x64.msi

安装步骤：

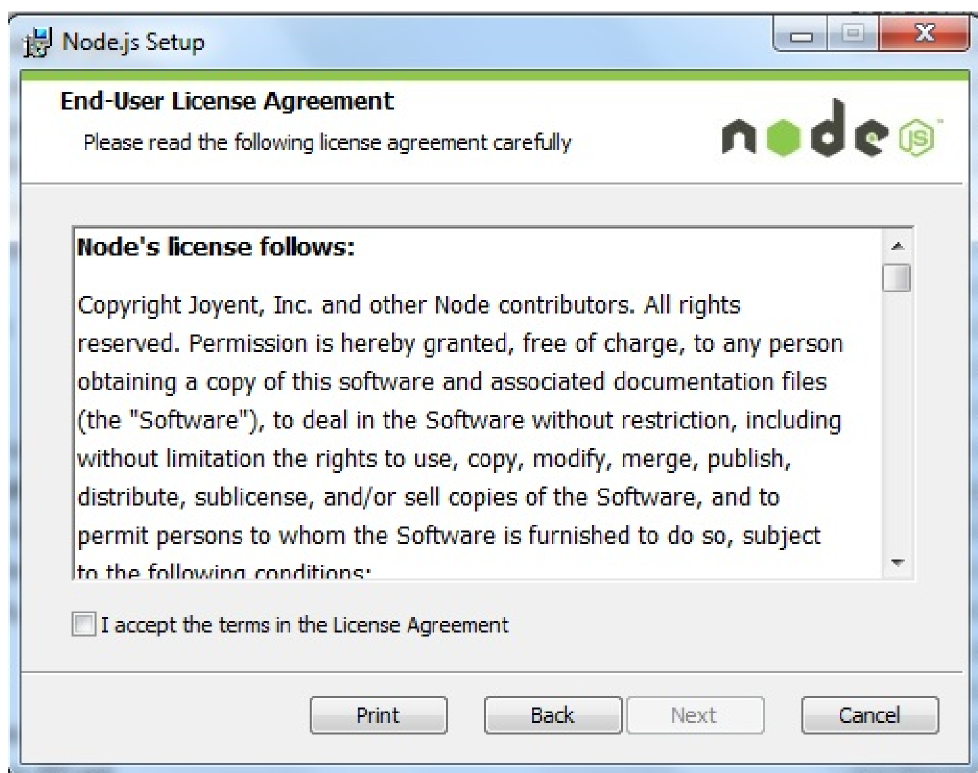
步骤 1 : 双击下载后的安装包 node-v0.10.26-x86.msi，如下所示：



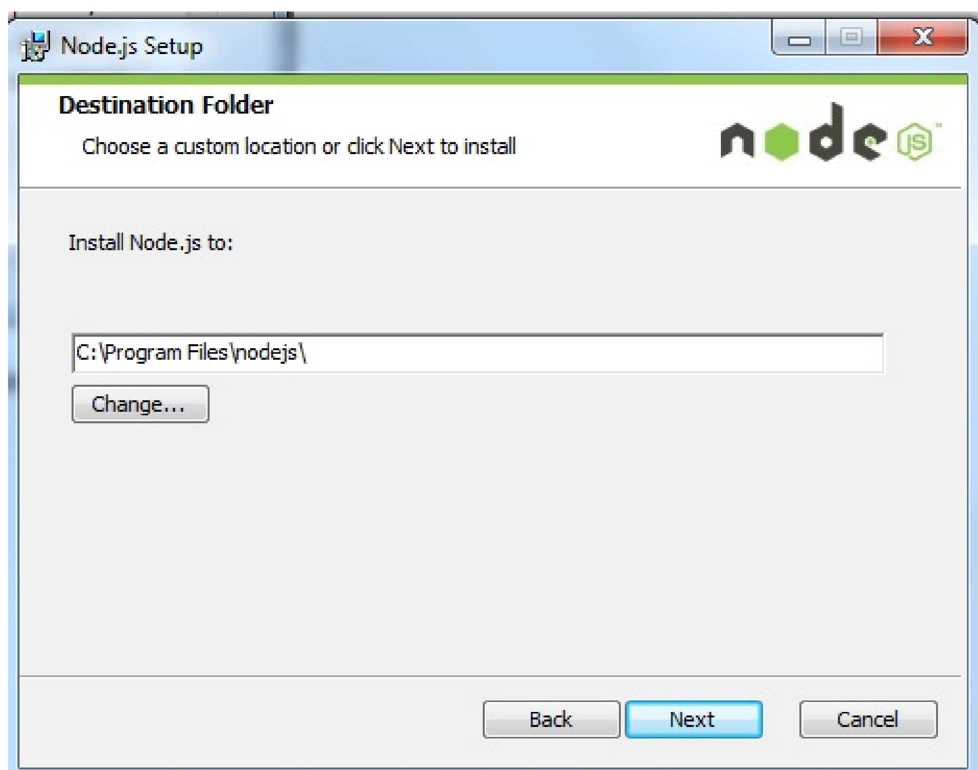
步骤 2 : 点击以上的Run(运行)，将出现如下界面：



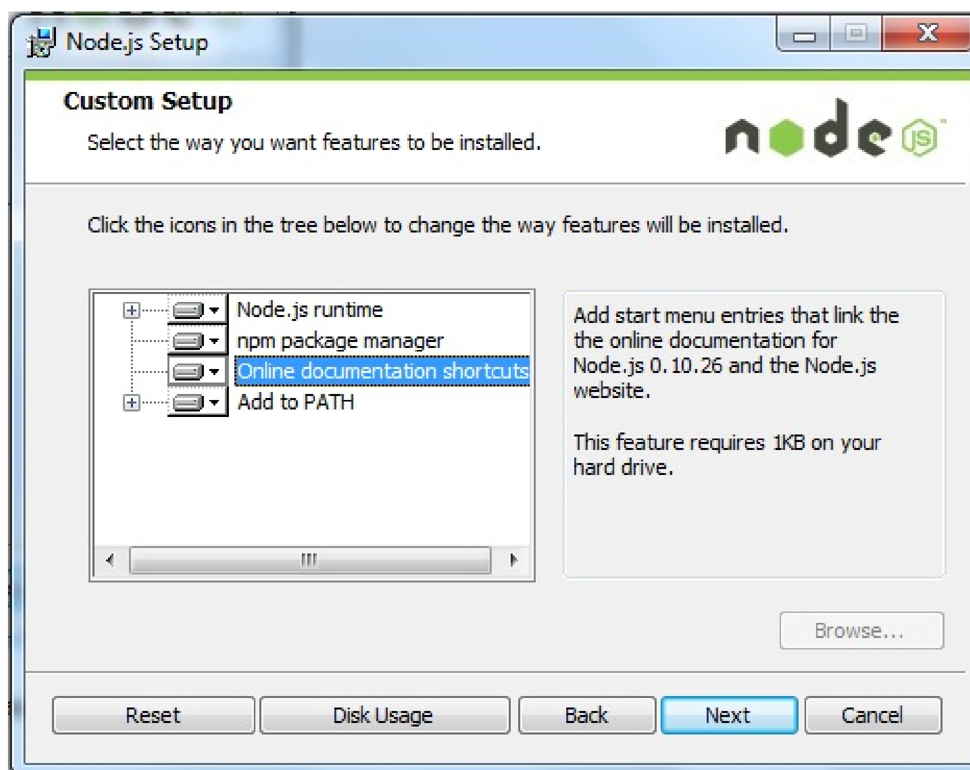
步骤 3 : 勾选接受协议选项，点击 next（下一步） 按钮 :



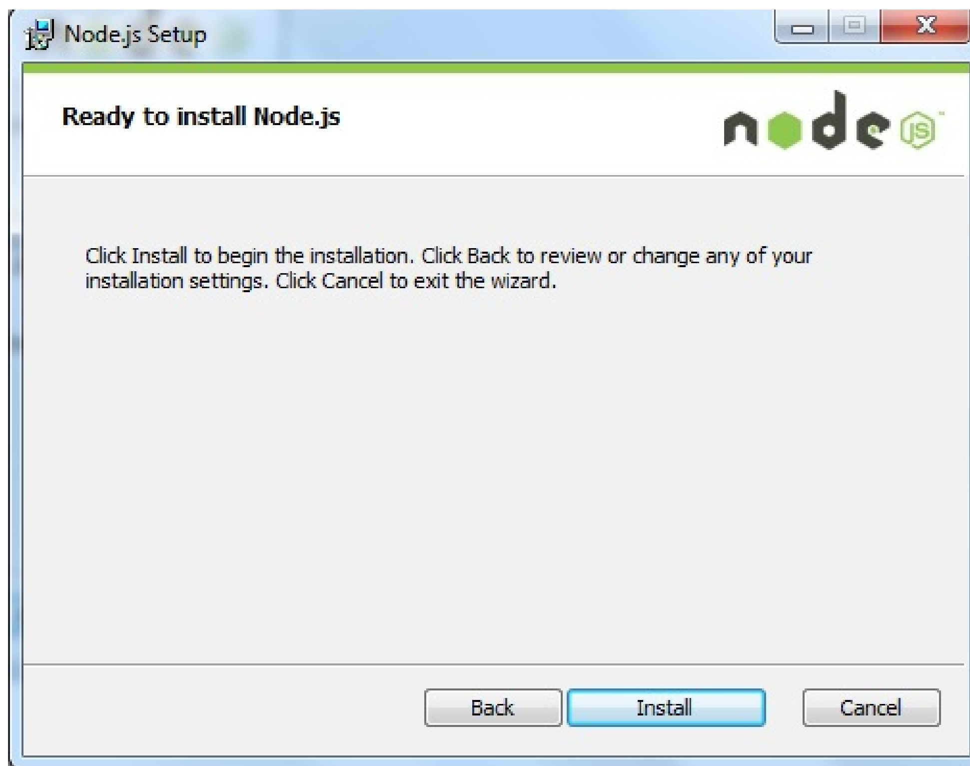
步骤 4 : Node.js默认安装目录为 "C:\Program Files\nodejs\" , 你可以修改目录， 并点击 next（下一步） ：



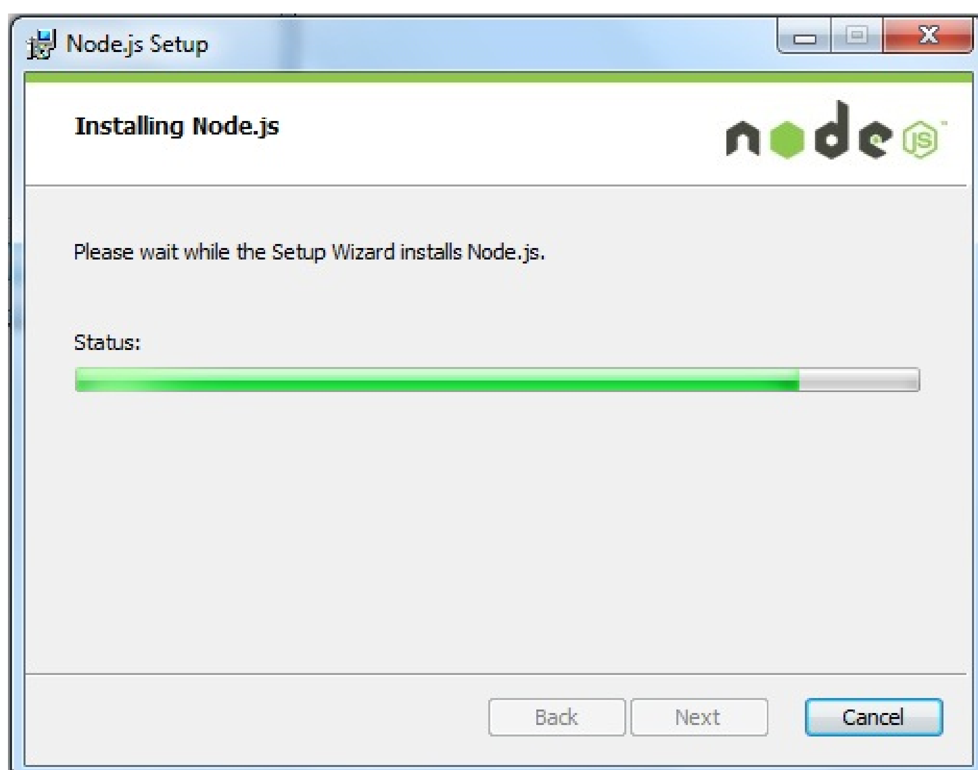
步骤 5 : 点击树形图标来选择你需要的安装模式 , 然后点击下一步 next（下一步）



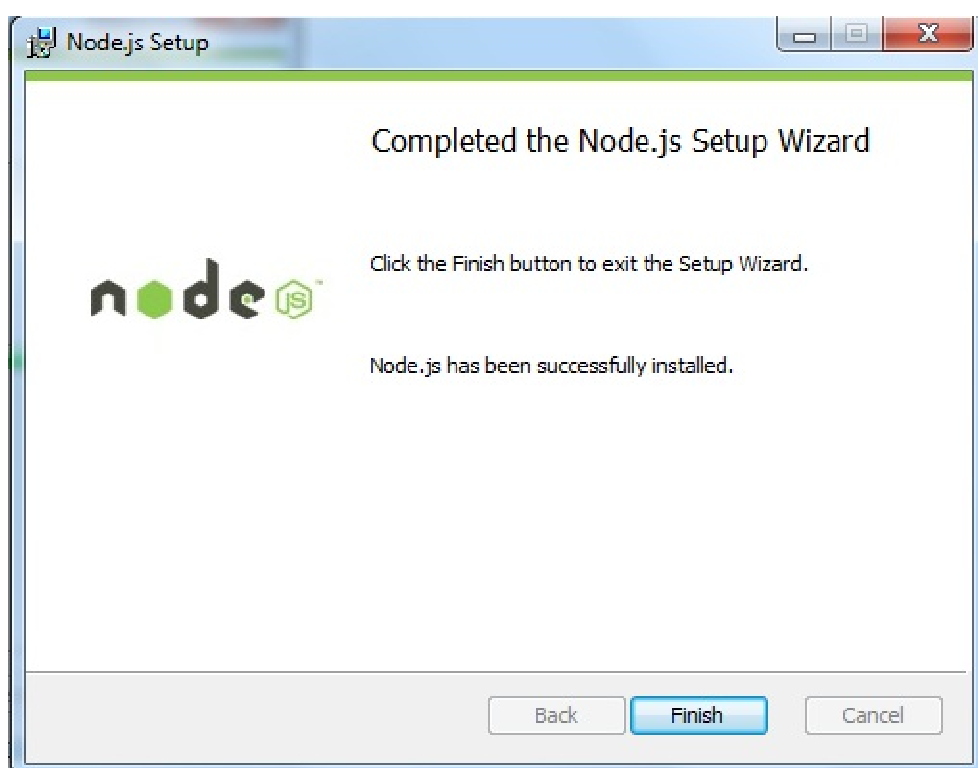
步骤 6 :点击 Install（安装） 开始安装Node.js。你也可以点击 Back（返回）来修改先前的配置。 然后并点击 next（下一步） ：



安装过程：



点击 Finish（完成）按钮退出安装向导。



检测PATH环境变量是否配置了Node.js，点击开始=》运行=》输入"cmd" => 输入命令"path"，输出如下结果：

```
PATH=C:\oraclexe\app\oracle\product\10.2.0\server\bin;C:\Windows\system32;
C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;
c:\python32\python;C:\MinGW\bin;C:\Program Files\GTK2-Runtime\lib;
C:\Program Files\MySQL\MySQL Server 5.5\bin;C:\Program Files\nodejs\;
C:\Users\rg\AppData\Roaming\npm
```

我们可以看到环境变量中已经包含了C:\Program Files\nodejs\

检查Node.js版本

```
E:\>node --version
v0.10.26

E:\>
```

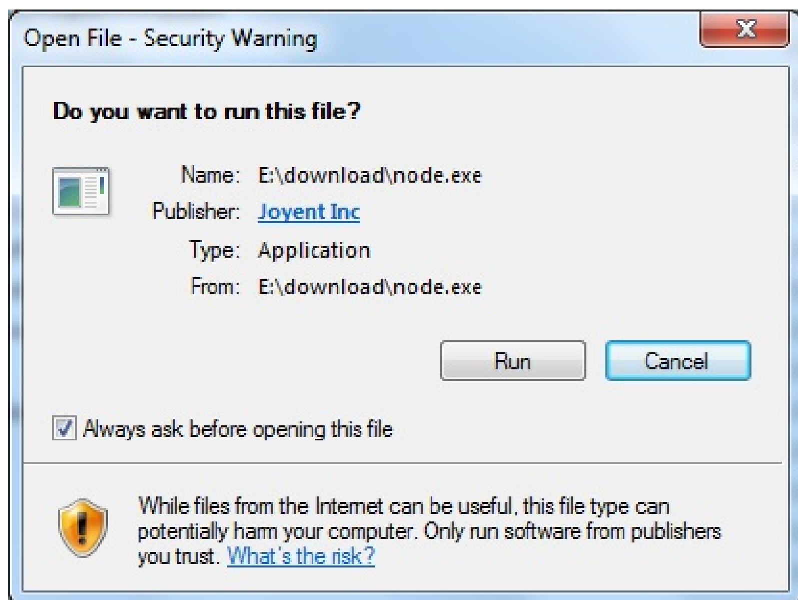
Windows 二进制文件 (.exe)安装：

32 位安装包下载地址：http://nodejs.org/dist/v0.10.26/node.exe

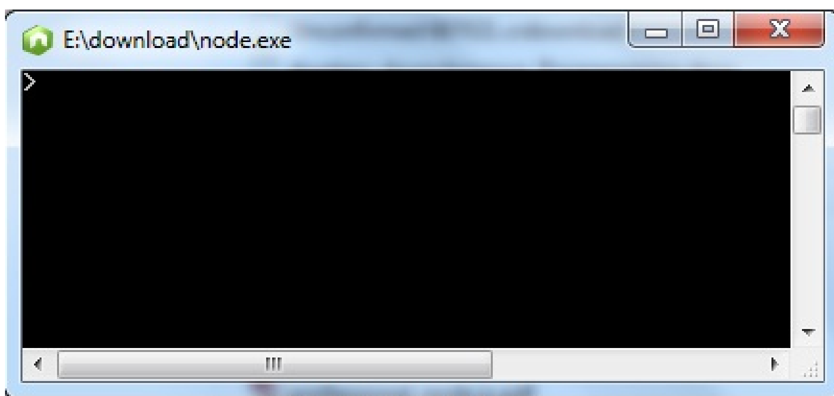
64 位安装包下载地址：http://nodejs.org/dist/v0.10.26/x64/node.exe

安装步骤

步骤 1：双击下载的安装包 Node.exe， 将出现如下界面：



点击 Run（运行）按钮将出现命令行窗口：



版本测试

进入 node.exe 所在的目录，如下所示：

```
E:\>cd download

E:\download>node --version
v0.10.26
```

如果你获得以上输出结果，说明你已经成功安装了Node.js。

Linux上安装 Node.js

Ubuntu 安装

以下部分我们将介绍在Ubuntu Linux下安装 Node.js 。 其他的Linux系统，如Centos等类似如下安装步骤。

在 Github 上获取 Node.js 源码：

```
ritwik@ritwik-pc:~$ sudo git clone https://github.com/joyent/node.git

ritwik@ritwik-pc:~$ sudo git clone https://github.com/joyent/node.git
Cloning into 'node'...
remote: Reusing existing pack: 121473, done.
remote: Counting objects: 49, done.
remote: Compressing objects: 100% (46/46), done.
Receiving objects:   8% (10851/121522), 2.84 MiB | 32 KiB/s
```

在完成下载后，将源码包名改为 'node'。

```
ritwik@ritwik-pc:~$ sudo git clone https://github.com/joyent/node.git
Cloning into 'node'...
remote: Reusing existing pack: 121473, done.
remote: Counting objects: 49, done.
remote: Compressing objects: 100% (46/46), done.
remote: Total 121522 (delta 13), reused 4 (delta 3)
Receiving objects: 100% (121522/121522), 91.16 MiB | 34 KiB/s, done.
Resolving deltas: 100% (91075/91075), done.
Checking out files: 100% (9604/9604), done.
ritwik@ritwik-pc:~$ 
```

修改目录权限：

```
ritwik@ritwik-pc:~$ sudo chmod 755 -R node
```

使用 './configure' 创建编译文件。

```
ritwik@ritwik-pc:~/node$ sudo ./configure
{ 'target_defaults': { 'cflags': [],
                       'default_configuration': 'Release',
                       'defines': ['OPENSSL_NO_SSL2=1'],
                       'include_dirs': [],
                       'libraries': []},
  'variables': { 'clang': 0,
                 'gcc_version': 47,
                 'host_arch': 'ia32',
                 'node_install_npm': 'true',
                 'node_prefix': '',
                 'node_shared_cares': 'false',
                 'node_shared_http_parser': 'false',
                 'node_shared_libuv': 'false',
                 'node_shared_openssl': 'false',
                 'node_shared_v8': 'false',
                 'node_shared_zlib': 'false',
                 'node_tag': '',
                 'node_use_dtrace': 'false',
                 'node_use_etw': 'false',
                 'node_use_mdb': 'false',
                 'node_use_openssl': 'true',
                 'node_use_perfctr': 'false',
                 'node_v8_options': '',
                 'python': '/usr/bin/python',
                 'target_arch': 'ia32',
                 'uv_library': 'static_library',
                 'uv_parent_path': '/deps/uv/',
                 'uv_use_dtrace': 'false',
                 'v8_enable_gdbjit': 0,
                 'v8_enable_i18n_support': 0,
                 'v8_no_strict_aliasing': 1,
                 'v8_optimized_debug': 0,
                 'v8_random_seed': 0,
                 'v8_use_snapshot': 'true'}}
creating  ./config.gypi
creating  ./config.mk
ritwik@ritwik-pc:~/node$ 
```

编译: make。

```
ritwik@ritwik-pc:~/node$ sudo make
make -C out BUILDTYPE=Release V=1
make[1]: Entering directory `/home/ritwik/node/out'
  cc '-DOPENSSL_NO_SSL2=1' '-D_DARWIN_USE_64_BIT_INODE=1' '-D_LARGEFILE_SOURCE' '-D_FILE_OFFSET_BITS=64' '-D_GNU_SOURCE' '-DHAVE_CONFIG_H' '-DCA
RES_STATICLIB' -I../deps/cares/include -I../deps/cares/src -I../deps/cares/config/linux  -pthread -Wall -Wextra -Wno-unused-parameter -m32 -g -p
edantic -Wall -Wextra -Wno-unused-parameter --std=gnu89 -O3 -ffunction-sections -fdata-sections -fno-tree-vrp -fno-omit-frame-pointer  -MMD -MF
/home/ritwik/node/out/Release/.deps//home/ritwik/node/out/Release/obj.target/cares/deps/cares/src/ares_cancel.o.d.raw  -c -o /home/ritwik/node/o
ut/Release/obj.target/cares/deps/cares/src/ares_cancel.o ../deps/cares/src/ares_cancel.c
  cc '-DOPENSSL_NO_SSL2=1' '-D_DARWIN_USE_64_BIT_INODE=1' '-D_LARGEFILE_SOURCE' '-D_FILE_OFFSET_BITS=64' '-D_GNU_SOURCE' '-DHAVE_CONFIG_H' '-DCA
RES_STATICLIB' -I../deps/cares/include -I../deps/cares/src -I../deps/cares/config/linux  -pthread -Wall -Wextra -Wno-unused-parameter -m32 -g -p
edantic -Wall -Wextra -Wno-unused-parameter --std=gnu89 -O3 -ffunction-sections -fdata-sections -fno-tree-vrp -fno-omit-frame-pointer  -MMD -MF
/home/ritwik/node/out/Release/.deps//home/ritwik/node/out/Release/obj.target/cares/deps/cares/src/ares__close_sockets.o.d.raw  -c -o /home/ritwi
k/node/out/Release/obj.target/cares/deps/cares/src/ares__close_sockets.o ../deps/cares/src/ares__close_sockets.c
  cc '-DOPENSSL_NO_SSL2=1' '-D_DARWIN_USE_64_BIT_INODE=1' '-D_LARGEFILE_SOURCE' '-D_FILE_OFFSET_BITS=64' '-D_GNU_SOURCE' '-DHAVE_CONFIG_H' '-DCA
RES_STATICLIB' -I../deps/cares/include -I../deps/cares/src -I../deps/cares/config/linux  -pthread -Wall -Wextra -Wno-unused-parameter -m32 -g -p
edantic -Wall -Wextra -Wno-unused-parameter --std=gnu89 -O3 -ffunction-sections -fdata-sections -fno-tree-vrp -fno-omit-frame-pointer  -MMD -MF
/home/ritwik/node/out/Release/.deps//home/ritwik/node/out/Release/obj.target/cares/deps/cares/src/ares_create_query.o.d.raw  -c -o /home/ritwik/
node/out/Release/obj.target/cares/deps/cares/src/ares_create_query.o ../deps/cares/src/ares_create_query.c
```

完成安装: make install。

```
ritwik@ritwik-pc:~/node$ sudo make install
```

最后我们输入'node --version' 命令来查看Node.js是否安装成功。

```
ritwik@ritwik-pc:~$ node --version
v0.11.13-pre
ritwik@ritwik-pc:~$
```

centOS下安装nodejs

1、下载源码，你需要在http://nodejs.org/下载最新的Nodejs版本，本文以v0.10.24为例:

```
cd /usr/local/src/
wget http://nodejs.org/dist/v0.10.24/node-v0.10.24.tar.gz
```

2、解压源码

```
tar zxvf node-v0.10.24.tar.gz
```

3、 编译安装

```
cd node-v0.10.24
./configure --prefix=/usr/local/node/0.10.24
make
make install
```

4、 配置NODE_HOME，进入profile编辑环境变量

```
vim /etc/profile
```

设置nodejs环境变量，在*export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL* 一行的上面添加如下内容:

```
#set for nodejs
export NODE_HOME=/usr/local/node/0.10.24
export PATH=$NODE_HOME/bin:$PATH
```

:wq保存并退出，编译/etc/profile 使配置生效

```
source /etc/profile
```

验证是否安装配置成功

```
node -v
```

输出 v0.10.24 表示配置成功

npm模块安装路径

```
/usr/local/node/0.10.24/lib/node_modules/
```

注：Nodejs 官网提供了编译好的Linux二进制包，你也可以下载下来直接应用。

Node.js 创建第一个应用

如果我们使用PHP来编写后端的代码时，需要Apache 或者 Nginx 的HTTP 服务器，并配上 mod_php5 模块和php-cgi。

从这个角度看，整个"接收 HTTP 请求并提供 Web 页面"的需求根本不需 要 PHP 来处理。

不过对 Node.js 来说，概念完全不一样了。使用 Node.js 时，我们不仅仅 在实现一个应用，同时还实现了整个 HTTP 服务器。事实上，我们的 Web 应用以及对应的 Web 服务器基本上是一样的。

在我们创建 Node.js 第一个 "Hello, World!" 应用前，让我们先了解下 Node.js 应用是由哪几部分组成的：

1. 引入 **required** 模块：我们可以使用 **require** 指令来载入 Node.js 模块。
2. 创建服务器：服务器可以监听客户端的请求，类似于 Apache 、Nginx 等 HTTP 服务器。
3. 接收请求与响应请求 服务器很容易创建，客户端可以使用浏览器或终端发送 HTTP 请求，服务器接收请求后返回响应数据。

创建 Node.js 应用

步骤一、引入 required 模块

我们使用 **require** 指令来载入 http 模块，并将实例化的 HTTP 赋值给变量 http，实例如下:

```
var http =  require("http");
```

步骤一、创建服务器

接下来我们使用 http.createServer() 方法创建服务器，并使用 listen 方法绑定 8888 端口。 函数通过 request, response 参数来接收和响应数据。

实例如下，在你项目的根目录下创建一个叫 server.js 的文件，并写入以下代码：

```
var http =  require('http'); http.createServer(function  (request, response)  {  // 发送 H
```

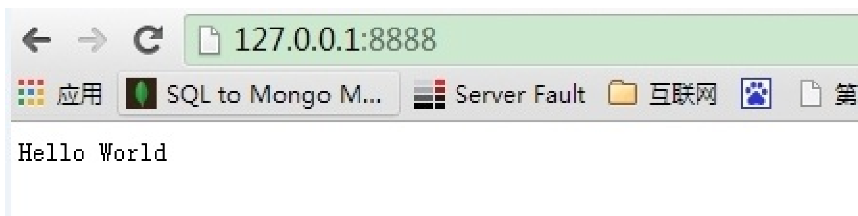
以上代码我们完成了一个可以工作的 HTTP 服务器。

使用 **node** 命令执行以上的代码：

```
node server.js Server running at http://127.0.0.1:8888/
```

```
E:\nodejs>node server.js
Server running at http://127.0.0.1:8888/
```

接下来，打开浏览器访问 http://127.0.0.1:8888/，你会看到一个写着 "Hello World"的网页。

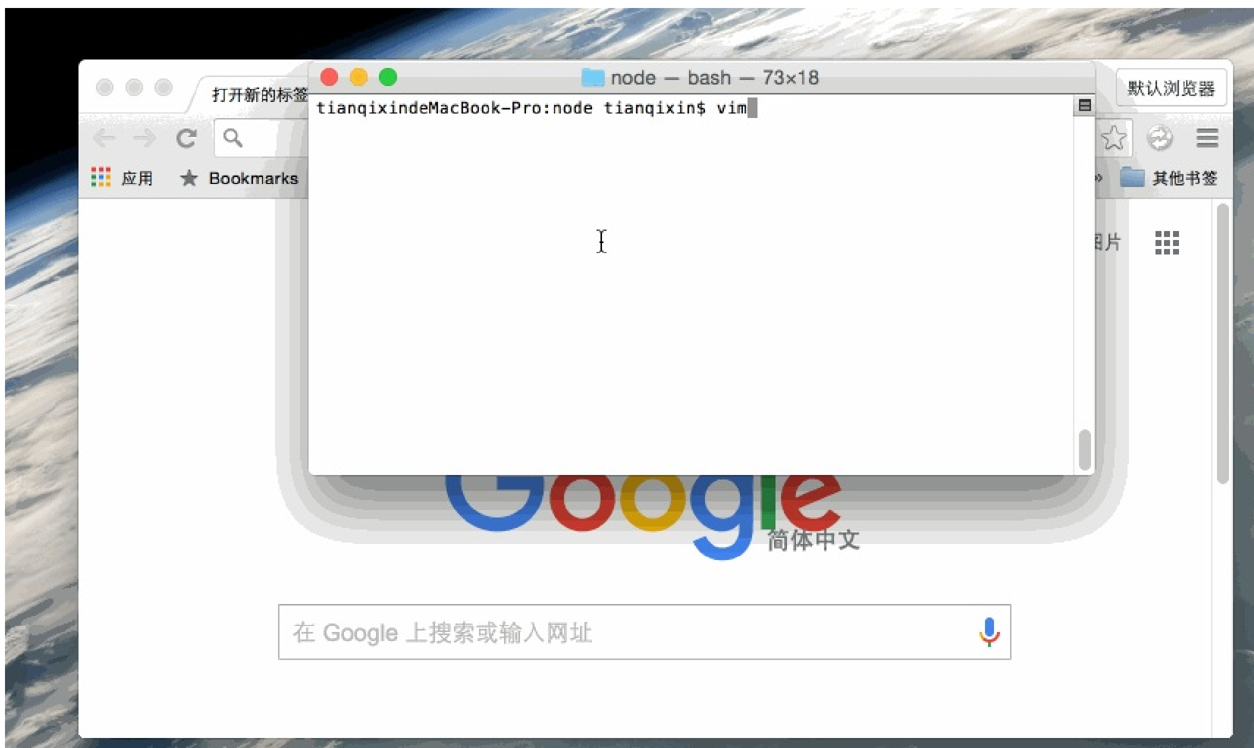


分析**Node.js** 的 **HTTP** 服务器：

- 第一行请求（require）Node.js 自带的 http 模块，并且把它赋值给 http 变量。
- 接下来我们调用 http 模块提供的函数： createServer 。这个函数会返回 一个对象，这个对象有一个叫做 listen 的方法，这个方法有一个数值参数， 指定这个 HTTP 服务器监听的端口号。

Gif 实例演示

接下来我们通过 Gif 图为大家演示实例操作：



NPM 使用介绍

NPM是随同NodeJS一起安装的包管理工具，能解决NodeJS代码部署上的很多问题，常见的使用场景有以下几种：

- 允许用户从NPM服务器下载别人编写的第三方包到本地使用。
- 允许用户从NPM服务器下载并安装别人编写的命令行程序到本地使用。
- 允许用户将自己编写的包或命令行程序上传到NPM服务器供别人使用。

由于新版的nodejs已经集成了npm，所以之前npm也一并安装好了。同样可以通过输入 **"npm -v"** 来测试是否成功安装。命令如下，出现版本提示表示安装成功:

```
$ npm -v
2.3.0
```

如果你安装的是旧版本的 npm，可以很容易得通过 npm 命令来升级，命令如下：

```
$ sudo npm install npm -g
/usr/local/bin/npm -> /usr/local/lib/node_modules/npm/bin/npm-cli.js
npm@2.14.2 /usr/local/lib/node_modules/npm
```

使用 npm 命令安装模块

npm 安装 Node.js 模块语法格式如下：

```
$ npm install <Module Name>
```

以下实例，我们使用 npm 命令安装常用的 Node.js web框架模块 **express**:

```
$ npm install express
```

安装好之后，express 包就放在了工程目录下的 node_modules 目录中，因此在代码中只需要通过 **require('express')** 的方式就好，无需指定第三方包路径。

```
var express = require('express');
```

全局安装与本地安装

npm 的包安装分为本地安装（local）、全局安装（global）两种，从敲的命令行来看，差别只是有没有-g而已，比如

```
npm install express          # 本地安装
npm install express -g   # 全局安装
```

如果出现以下错误：

```
npm err! Error: connect ECONNREFUSED 127.0.0.1:8087
```

解决办法为：

```
$ npm config set proxy null
```

本地安装

- 1. 将安装包放在 ./node_modules 下（运行 npm 命令时所在的目录），如果没有 node_modules 目录，会在当前执行 npm 命令的目录下生成 node_modules 目录。
- 2. 可以通过 require() 来引入本地安装的包。

全局安装

- 1. 将安装包放在 /usr/local 下。
- 2. 可以直接在命令行里使用。
- 3. 不能通过 require() 来引入本地安装的包。

接下来我们使用全局方式安装 express

```
$ npm install express -g
```

安装过程输出如下内容，第一行输出了模块的版本号及安装位置。

```
express@4.13.3 node_modules/express
├── escape-html@1.0.2
├── range-parser@1.0.2
├── merge-descriptors@1.0.0
├── array-flatten@1.1.1
├── cookie@0.1.3
├── utils-merge@1.0.0
├── parseurl@1.3.0
├── cookie-signature@1.0.6
├── methods@1.1.1
├── fresh@0.3.0
├── vary@1.0.1
├── path-to-regexp@0.1.7
├── content-type@1.0.1
├── etag@1.7.0
├── serve-static@1.10.0
├── content-disposition@0.5.0
├── depd@1.0.1
├── qs@4.0.0
├── finalhandler@0.4.0 (unpipe@1.0.0)
├── on-finished@2.3.0 (ee-first@1.1.1)
├── proxy-addr@1.0.8 (forwarded@0.1.0, ipaddr.js@1.0.1)
├── debug@2.2.0 (ms@0.7.1)
├── type-is@1.6.8 (media-typer@0.3.0, mime-types@2.1.6)
├── accepts@1.2.12 (negotiator@0.5.3, mime-types@2.1.6)
└── send@0.13.0 (destroy@1.0.3, statuses@1.2.1, ms@0.7.1, mime@1.3.4, http-errors@1.3.1)
```

你可以使用以下命令来查看所有全局安装的模块：

```
$ npm ls -g
```

使用 package.json

package.json 位于模块的目录下，用于定义包的属性。接下来让我们来看下 express 包的 package.json 文件，位于 node_modules/express/package.json 内容：

```
{
  "name": "express",
  "description": "Fast, unopinionated, minimalist web framework",
  "version": "4.13.3",
  "author": {
    "name": "TJ Holowaychuk",
    "email": "tj@vision-media.ca"
  },
  "contributors": [
    {
      "name": "Aaron Heckmann",
      "email": "aaron.heckmann+github@gmail.com"
    },
    {
      "name": "Ciaran Jessup",
      "email": "ciaranj@gmail.com"
    },
    {
      "name": "Douglas Christopher Wilson",
      "email": "doug@somethingdoug.com"
    },
    {
      "name": "Guillermo Rauch",
      "email": "rauchg@gmail.com"
```

```
    },
    {
      "name": "Jonathan Ong",
      "email": "me@jongleberry.com"
    },
    {
      "name": "Roman Shtylman",
      "email": "shtylman+expressjs@gmail.com"
    },
    {
      "name": "Young Jae Sim",
      "email": "hanul@hanul.me"
    }
  ],
  "license": "MIT",
  "repository": {
    "type": "git",
    "url": "git+https://github.com/strongloop/express.git"
  },
  "homepage": "http://expressjs.com/",
  "keywords": [
    "express",
    "framework",
    "sinatra",
    "web",
    "rest",
    "restful",
    "router",
    "app",
    "api"
  ],
  "dependencies": {
    "accepts": "~1.2.12",
    "array-flatten": "1.1.1",
    "content-disposition": "0.5.0",
    "content-type": "~1.0.1",
    "cookie": "0.1.3",
    "cookie-signature": "1.0.6",
    "debug": "~2.2.0",
    "depd": "~1.0.1",
    "escape-html": "1.0.2",
    "etag": "~1.7.0",
    "finalhandler": "0.4.0",
    "fresh": "0.3.0",
    "merge-descriptors": "1.0.0",
    "methods": "~1.1.1",
    "on-finished": "~2.3.0",
    "parseurl": "~1.3.0",
    "path-to-regexp": "0.1.7",
    "proxy-addr": "~1.0.8",
    "qs": "4.0.0",
    "range-parser": "~1.0.2",
    "send": "0.13.0",
    "serve-static": "~1.10.0",
    "type-is": "~1.6.6",
    "utils-merge": "1.0.0",
    "vary": "~1.0.1"
  },
  "devDependencies": {
    "after": "0.8.1",
    "ejs": "2.3.3",
    "istanbul": "0.3.17",
    "marked": "0.3.5",
    "mocha": "2.2.5",
    "should": "7.0.2",
    "supertest": "1.0.1",
    "body-parser": "~1.13.3",
    "connect-redis": "~2.4.1",
    "cookie-parser": "~1.3.5",
    "cookie-session": "~1.2.0",
    "express-session": "~1.11.3",
    "jade": "~1.11.0",
```

```
    "method-override": "~2.3.5",
    "morgan": "~1.6.1",
    "multiparty": "~4.1.2",
    "vhost": "~3.0.1"
  },
  "engines": {
    "node": ">= 0.10.0"
  },
  "files": [
    "LICENSE",
    "History.md",
    "Readme.md",
    "index.js",
    "lib/"
  ],
  "scripts": {
    "test": "mocha --require test/support/env --reporter spec --bail --check-leaks test/
    "test-ci": "istanbul cover node_modules/mocha/bin/_mocha --report lcovonly -- --requi
    "test-cov": "istanbul cover node_modules/mocha/bin/_mocha -- --require test/support/e
    "test-tap": "mocha --require test/support/env --reporter tap --check-leaks test/ test
  },
  "gitHead": "ef7ad681b245fba023843ce94f6bcb8e275bbb8e",
  "bugs": {
    "url": "https://github.com/strongloop/express/issues"
  },
  "_id": "express@4.13.3",
  "_shasum": "ddb2f1fb4502bf33598d2b032b037960ca6c80a3",
  "_from": "express@*",
  "_npmVersion": "1.4.28",
  "_npmUser": {
    "name": "dougwilson",
    "email": "doug@somethingdoug.com"
  },
  "maintainers": [
    {
      "name": "tjholowaychuk",
      "email": "tj@vision-media.ca"
    },
    {
      "name": "jongleberry",
      "email": "jonathanrichardong@gmail.com"
    },
    {
      "name": "dougwilson",
      "email": "doug@somethingdoug.com"
    },
    {
      "name": "rfeng",
      "email": "enjoyjava@gmail.com"
    },
    {
      "name": "aredridel",
      "email": "aredridel@dinhe.net"
    },
    {
      "name": "strongloop",
      "email": "callback@strongloop.com"
    },
    {
      "name": "defunctzombie",
      "email": "shtylman@gmail.com"
    }
  ],
  "dist": {
    "shasum": "ddb2f1fb4502bf33598d2b032b037960ca6c80a3",
    "tarball": "http://registry.npmjs.org/express/-/express-4.13.3.tgz"
  },
  "directories": {},
  "_resolved": "https://registry.npmjs.org/express/-/express-4.13.3.tgz",
  "readme": "ERROR: No README data found!"
}
```



Package.json 属性说明

- **name** - 包名。
- **version** - 包的版本号。
- **description** - 包的描述。
- **homepage** - 包的官网 url 。
- **author** - 包的作者姓名。
- **contributors** - 包的其他贡献者姓名。
- **dependencies** - 依赖包列表。如果依赖包没有安装，npm 会自动将依赖包安装在 node_module 目录下。
- **repository** - 包代码存放的地方的类型，可以是 git 或 svn，git 可在 Github 上。
- **main** - main 字段是一个模块ID，它是一个指向你程序的主要项目。就是说，如果你包的名字叫 express，然后用户安装它，然后require("express")。
- **keywords** - 关键字

卸载模块

我们可以使用以下命令来卸载 Node.js 模块。

```
$ npm uninstall express
```

卸载后，你可以到 /node_modules/ 目录下查看包是否还存在，或者使用以下命令查看：

```
$ npm ls
```

更新模块

我们可以使用以下命令更新模块：

```
$ npm update express
```

搜索模块

使用以下来搜索模块：

```
$ npm search express
```

创建模块

创建模块，package.json 文件是必不可少的。我们可以使用 NPM 生成 package.json 文件，生成的文件包含了基本的结果。

```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (node_modules) runoob                   # 模块名
version: (1.0.0)
description: Node.js 测试模块(www.runoob.com)  # 描述
entry point: (index.js)
test command: make test
git repository: https://github.com/runoob/runoob.git  # Github 地址
keywords:
author:
license: (ISC)
About to write to ……/node_modules/package.json:      # 生成地址

{
  "name": "runoob",
  "version": "1.0.0",
  "description": "Node.js 测试模块(www.runoob.com)",
  ……
}

Is this ok? (yes) yes
```

以上的信息，你需要根据你自己的情况输入。在最后输入 "yes" 后会生成 package.json 文件。

接下来我们可以使用以下命令在 npm 资源库中注册用户（使用邮箱注册）：

```
$ npm adduser
Username: mcmohd
Password:
Email: (this IS public) mcmohd@gmail.com
```

接下来我们就用以下命令来发布模块：

```
$ npm publish
```

如果你以上的步骤都操作正确，你就可以跟其他模块一样使用 npm 来安装。

版本号

使用NPM下载和发布代码时都会接触到版本号。NPM使用语义版本号来管理代码，这里简单介绍一下。

语义版本号分为X.Y.Z三位，分别代表主版本号、次版本号和补丁版本号。当代码变更时，版本号按以下原则更新。

- 如果只是修复bug，需要更新Z位。
- 如果是新增了功能，但是向下兼容，需要更新Y位。
- 如果有大变动，向下不兼容，需要更新X位。

版本号有了这个保证后，在申明第三方包依赖时，除了可依赖于一个固定版本号外，还可依赖于某个范围的版本号。例如"argv": "0.0.x"表示依赖于0.0.x系列的最新版argv。

NPM支持的所有版本号范围指定方式可以查看官方文档。

NPM 常用命令

除了本章介绍的部分外，NPM还提供了很多功能，package.json里也有很多其它有用的字段。

除了可以在npmjs.org/doc/查看官方文档外，这里再介绍一些NPM常用命令。

NPM提供了很多命令，例如install和publish，使用npm help可查看所有命令。

- NPM提供了很多命令，例如 `install` 和 `publish` ，使用 `npm help` 可查看所有命令。
- 使用 `npm help <command>` 可查看某条命令的详细帮助，例如 `npm help install` 。
- 在 `package.json` 所在目录下使用 `npm install . -g` 可先在本地安装当前命令行程序，可用于发布前的本地测试。
- 使用 `npm update <package>` 可以把当前目录下 `node_modules` 子目录里边的对应模块更新至最新版本。
- 使用 `npm update <package> -g` 可以把全局安装的对应命令行程序更新至最新版。
- 使用 `npm cache clear` 可以清空NPM本地缓存，用于对付使用相同版本号发布新版本代码的人。
- 使用 `npm unpublish <package>@<version>` 可以撤销发布自己发布过的某个版本代码。

Node.js REPL(交互式解释器)

Node.js REPL(Read Eval Print Loop:交互式解释器) 表示一个电脑的环境，类似 Window 系统的终端或 Unix/Linux shell，我们可以在终端中输入命令，并接收系统的响应。

Node 自带了交互式解释器，可以执行以下任务：

- 读取 - 读取用户输入，解析输入了Javascript 数据结构并存储在内存中。
- 执行 - 执行输入的数据结构
- 打印 - 输出结果
- 循环 - 循环操作以上步骤直到用户两次按下 **ctrl-c** 按钮退出。

Node 的交互式解释器可以很好的调试 Javascript 代码。

开始学习 REPL

我们可以输入以下命令来启动 Node 的终端：

```
$ node
>
```

这时我们就可以在 > 后输入简单的表达式，并按下回车键来计算结果。

简单的表达式运算

接下来让我们在 Node.js REPL 的命令行窗口中执行简单的数学运算：

```
$ node
> 1 +4
5
> 5 / 2
2.5
> 3 * 6
18
> 4 - 1
3
> 1 + ( 2 * 3 ) - 4
3
>
```

使用变量

你可以将数据存储在变量中，并在你需要的使用它。

变量声明需要使用 **var** 关键字，如果没有使用 var 关键字变量会直接打印出来。

使用 **var** 关键字的变量可以使用 console.log() 来输出变量。

```
$ node
> x = 10
10
> var y = 10
undefined
> x + y
20
> console.log("Hello World")
Hello World
undefined
> console.log("www.runoob.com")
www.runoob.com
undefined
```

多行表达式

Node REPL 支持输入多行表达式，这就有点类似 JavaScript。接下来让我们来执行一个 do-while 循环：

```
$ node
> var x = 0
undefined
> do {
... x++;
... console.log("x: " + x);
... } while ( x < 5 );
x: 1
x: 2
x: 3
x: 4
x: 5
undefined
>
```

... 三个点的符号是系统自动生成的，你回车换行后即可。Node 会自动检测是否为连续的表达式。

下划线(_)变量

你可以使用下划线(_)获取表达式的运算结果：

```
$ node
> var x = 10
undefined
> var y = 20
undefined
> x + y
30
> var sum = _
undefined
> console.log(sum)
30
undefined
>
```

REPL 命令

- **ctrl + c** - 退出当前终端。
- **ctrl + c 按下两次** - 退出 Node REPL。
- **ctrl + d** - 退出 Node REPL.
- **向上/向下 键** - 查看输入的历史命令
- **tab 键** - 列出当前命令
- **.help** - 列出使用命令
- **.break** - 退出多行表达式
- **.clear** - 退出多行表达式
- **.save *filename*** - 保存当前的 Node REPL 会话到指定文件
- **.load *filename*** - 载入当前 Node REPL 会话的文件内容。

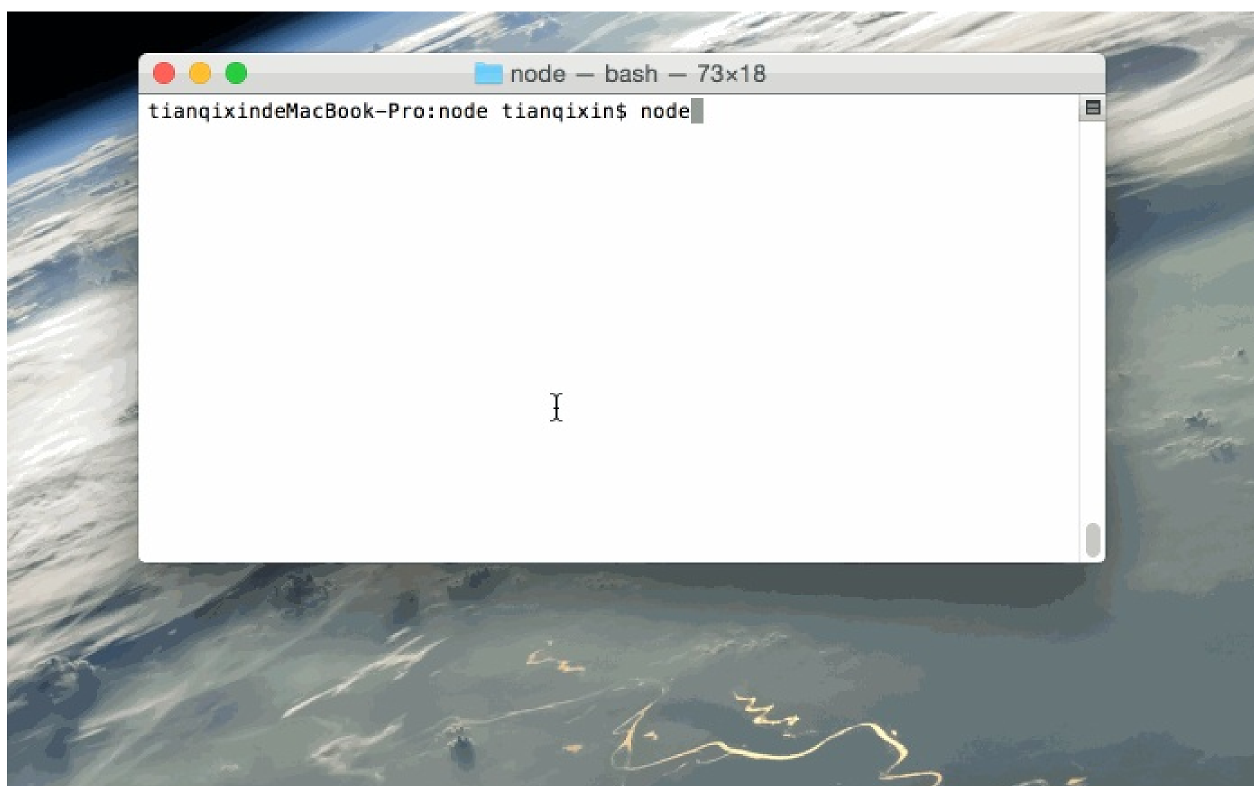
停止 REPL

前面我们已经提到按下两次 **ctrl + c** 建就能退出 REPL:

```
$ node
>
(^C again to quit)
>
```

Gif 实例演示

接下来我们通过 Gif 图为大家演示实例操作：


node — bash — 73×18
tianqixindeMacBook-Pro:node tianqixin$ node

Node.js 回调函数

Node.js 异步编程的直接体现就是回调。

异步编程依托于回调来实现，但不能说使用了回调后程序就异步化了。

回调函数在完成任务后就会被调用，Node 使用了大量的回调函数，Node 所有 API 都支持回调函数。

例如，我们可以一边读取文件，一边执行其他命令，在文件读取完成后，我们将文件内容作为回调函数的参数返回。这样在执行代码时就没有阻塞或等待文件 I/O 操作。这就大大提高了 Node.js 的性能，可以处理大量的并发请求。

阻塞代码实例

创建一个文件 input.txt ，内容如下：

```
菜鸟教程官网地址：www.runoob.com
```

创建 main.js 文件, 代码如下：

```
var fs = require("fs");

var data = fs.readFileSync('input.txt');

console.log(data.toString());
console.log("程序执行结束!");
```

以上代码执行结果如下：

```
$ node main.js
菜鸟教程官网地址：www.runoob.com

程序执行结束!
```

非阻塞代码实例

创建一个文件 input.txt ，内容如下：

```
菜鸟教程官网地址：www.runoob.com
```

创建 main.js 文件, 代码如下：

```
var fs = require("fs");

fs.readFile('input.txt', function (err, data) {
    if (err) return console.error(err);
    console.log(data.toString());
});

console.log("程序执行结束!");
```

以上代码执行结果如下：

```
$ node main.js
程序执行结束!
菜鸟教程官网地址：www.runoob.com
```

以上两个实例我们了解了阻塞与非阻塞调用的不同。第一个实例在文件读取完后才执行完程序。 第二个实例我们呢不需要等待文件读取完，这样就可以在读取文件时同时执行接下来的代码，大大提高了程序的性能。

因此，阻塞按是按顺序执行的，而非阻塞是不需要按顺序的，所以如果需要处理回调函数的参数，我们就需要写在回调函数内。

Node.js 事件循环

Node.js 是单进程单线程应用程序，但是通过事件和回调支持并发，所以性能非常高。

Node.js 的每一个 API 都是异步的，并作为一个独立线程运行，使用异步函数调用，并处理并发。

Node.js 基本上所有的事件机制都是用设计模式中观察者模式实现。

Node.js 单线程类似进入一个while(true)的事件循环，直到没有事件观察者退出，每个异步事件都生成一个事件观察者，如果有事件发生就调用该回调函数.

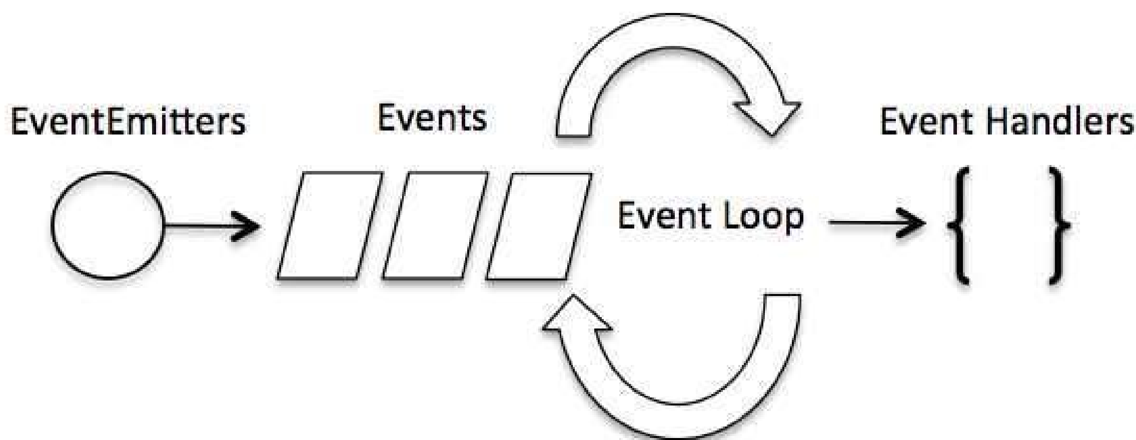
事件驱动程序

Node.js 使用事件驱动模型，当web server接收到请求，就把它关闭然后进行处理，然后去服务下一个web请求。

当这个请求完成，它被放回处理队列，当到达队列开头，这个结果被返回给用户。

这个模型非常高效可扩展性非常强，因为webserver一直接受请求而不等待任何读写操作。（这也被称之为非阻塞式IO或者事件驱动IO）

在事件驱动模型中，会生成一个主循环来监听事件，当检测到事件时触发回调函数。



整个事件驱动的流程就是这么实现的，非常简洁。有点类似于观察者模式，事件相当于一个主题(Subject)，而所有注册到这个事件上的处理函数相当于观察者(Observer)。

Node.js 有多个内置的事件，我们可以通过引入 events 模块，并通过实例化 EventEmitter 类来绑定和监听事件，如下实例：

```
// 引入 events 模块
var events = require('events');
// 创建 eventEmitter 对象
var eventEmitter = new events.EventEmitter();
```

以下程序绑定事件处理程序：

```
// 绑定事件及事件的处理程序
eventEmitter.on('eventName', eventHandler);
```

我们可以通过程序触发事件：

```
// 触发事件
eventEmitter.emit('eventName');
```

实例

创建 main.js 文件，代码如下所示：

```
// 引入 events 模块
var events = require('events');
// 创建 eventEmitter 对象
var eventEmitter = new events.EventEmitter();

// 创建事件处理程序
var connectHandler = function connected() {
   console.log('连接成功。');

   // 触发 data_received 事件
   eventEmitter.emit('data_received');
}

// 绑定 connection 事件处理程序
eventEmitter.on('connection', connectHandler);

// 使用匿名函数绑定 data_received 事件
eventEmitter.on('data_received', function(){
   console.log('数据接收成功。');
});

// 触发 connection 事件
eventEmitter.emit('connection');

console.log("程序执行完毕。");
```

接下来让我们执行以上代码：

```
$ node main.js
连接成功。
数据接收成功。
程序执行完毕。
```

Node 应用程序是如何工作的？

在 Node 应用程序中，执行异步操作的函数将回调函数作为最后一个参数， 回调函数接收错误对象作为第一个参数。

接下来让我们来重新看下前面的实例，创建一个 input.txt ,文件内容如下：

```
菜鸟教程官网地址：www.runoob.com
```

创建 main.js 文件，代码如下：

```
var fs = require("fs");

fs.readFile('input.txt', function (err, data) {
   if (err){
      console.log(err.stack);
      return;
   }
   console.log(data.toString());
});
console.log("程序执行完毕");
```

以上程序中 fs.readFile() 是异步函数用于读取文件。 如果在读取文件过程中发生错误，错误 err 对象就会输出错误信息。

如果没发生错误，readFile 跳过 err 对象的输出，文件内容就通过回调函数输出。

执行以上代码，执行结果如下：

```
程序执行完毕
菜鸟教程官网地址：www.runoob.com
```

接下来我们删除 input.txt 文件，执行结果如下所示：

```
程序执行完毕
Error: ENOENT, open 'input.txt'
```

因为文件 input.txt 不存在，所以输出了错误信息。

Node.js EventEmitter

Node.js 所有的异步 I/O 操作在完成时都会发送一个事件到事件队列。

Node.js里面的许多对象都会分发事件：一个net.Server对象会在每次有新连接时分发一个事件， 一个fs.readStream对象会在文件被打开的时候发出一个事件。 所有这些产生事件的对象都是 events.EventEmitter 的实例。

EventEmitter 类

events 模块只提供了一个对象： events.EventEmitter。EventEmitter 的核心就是事件触发与事件监听器功能的封装。

你可以通过require("events");来访问该模块。

```
// 引入 events 模块
var events = require('events');
// 创建 eventEmitter 对象
var eventEmitter = new events.EventEmitter();
```

EventEmitter 对象如果在实例化时发生错误，会触发 'error' 事件。当添加新的监听器时，'newListener' 事件会触发，当监听器被移除时，'removeListener' 事件被触发。

下面我们用一个简单的例子说明 EventEmitter 的用法：

```
//event.js 文件
var EventEmitter = require('events').EventEmitter;
var event = new EventEmitter();
event.on('some_event', function() {
    console.log('some_event 事件触发');
});
setTimeout(function() {
    event.emit('some_event');
}, 1000);
```

执行结果如下：

运行这段代码，1 秒后控制台输出了 **'some_event 事件触发'**。其原理是 event 对象注册了事件 some_event 的一个监听器，然后我们通过 setTimeout 在 1000 毫秒以后向 event 对象发送事件 some_event，此时会调用some_event 的监听器。

```
$ node event.js
some_event 事件触发
```

EventEmitter 的每个事件由一个事件名和若干个参数组成，事件名是一个字符串，通常表达一定的语义。对于每个事件，EventEmitter 支持 若干个事件监听器。

当事件触发时，注册到这个事件的事件监听器被依次调用，事件参数作为回调函数参数传递。

让我们以下面的例子解释这个过程：

```
//event.js 文件
var events = require('events');
var emitter = new events.EventEmitter();
emitter.on('someEvent', function(arg1, arg2) {
    console.log('listener1', arg1, arg2);
});
emitter.on('someEvent', function(arg1, arg2) {
    console.log('listener2', arg1, arg2);
});
emitter.emit('someEvent', 'arg1 参数', 'arg2 参数');
```

执行以上代码，运行的结果如下：/p>

```
$ node event.js
listener1 arg1 参数 arg2 参数
listener2 arg1 参数 arg2 参数
```

以上例子中，emitter 为事件 someEvent 注册了两个事件监听器，然后触发了 someEvent 事件。

运行结果中可以看到两个事件监听器回调函数被先后调用。 这就是EventEmitter最简单的用法。

EventEmitter 提供了多个属性，如 **on** 和 **emit**。**on** 函数用于绑定事件函数，**emit** 属性用于触发一个事件。接下来我们来具体看下 EventEmitter 的属性介绍。

方法

方法	
addListener(event, listener)	为指定事件添加一个监听器到监听器数组的尾部。
on(event, listener)	为指定事件注册一个监听器，接受一个字符串 event 和一
once(event, listener)	为指定事件注册一个单次监听器，即 监听器最多只会触发 `server.once('connection', function (stream) { console.l`
removeListener(event, listener)	移除指定事件的某个监听器，监听器 必须是该事件已经注 `var callback = function(stream) { console.log('someone`
removeAllListeners([event])	移除所有事件的所有监听器， 如果指定事件，则移除指定
setMaxListeners(n)	默认情况下， EventEmitters 如果你添加的监听器超过 10
listeners(event)	返回指定事件的监听器数组。
emit(event, [arg1], [arg2], [...])	按参数的顺序执行每个监听器，如果事件有注册监听返回

类方法

方法	描述
listenerCount(emitter, event)	返回指定事件的监听器数量。

事件

事件	描述
newListener	**event** - 字符串，事件名称 **listener** - 处理事件函数 该事件在添加新监听器时被触发。
removeListener	**event** - 字符串，事件名称 **listener** - 处理事件函数 从指定监听器数组中删除一个监听器。需要注意的是，此操作将会改变处于被删监听器之后的那些监听器的索引。

实例

以下实例通过 connection（连接）事件演示了 EventEmitter 类的应用。

创建 main.js 文件，代码如下：

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

// 监听器 #1
var listner1 = function listner1() {
   console.log('监听器 listner1 执行。');
}

// 监听器 #2
var listner2 = function listner2() {
  console.log('监听器 listner2 执行。');
}

// 绑定 connection 事件，处理函数为 listner1
eventEmitter.addListener('connection', listner1);

// 绑定 connection 事件，处理函数为 listner2
eventEmitter.on('connection', listner2);

var eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connectio
console.log(eventListeners + " 监听器监听连接事件。");

// 处理 connection 事件
eventEmitter.emit('connection');

// 移除监绑定的 listner1 函数
eventEmitter.removeListener('connection', listner1);
console.log("listner1 不再受监听。");

// 触发连接事件
eventEmitter.emit('connection');

eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connection');
console.log(eventListeners + " 监听器监听连接事件。");

console.log("程序执行完毕。");
```

以上代码，执行结果如下所示：

```
$ node main.js
2 监听器监听连接事件。
监听器 listner1 执行。
监听器 listner2 执行。
listner1 不再受监听。
监听器 listner2 执行。
1 监听器监听连接事件。
程序执行完毕。
```

error 事件

EventEmitter 定义了一个特殊的事件 error，它包含了错误的语义，我们在遇到 异常的时候通常会触发 error 事件。

当 error 被触发时，EventEmitter 规定如果没有响 应的监听器，Node.js 会把它当作异常，退出程序并输出错误信息。

我们一般要为会触发 error 事件的对象设置监听器，避免遇到错误后整个程序崩溃。例如：

```
var events = require('events');
var emitter = new events.EventEmitter();
emitter.emit('error');
```

运行时会显示以下错误：

```
node.js:201
throw e; // process.nextTick error, or 'error' event on first tick
^
Error: Uncaught, unspecified 'error' event.
at EventEmitter.emit (events.js:50:15)
at Object.<anonymous> (/home/byvoid/error.js:5:9)
at Module._compile (module.js:441:26)
at Object..js (module.js:459:10)
at Module.load (module.js:348:31)
at Function._load (module.js:308:12)
at Array.0 (module.js:479:10)
at EventEmitter._tickCallback (node.js:192:40)
```

继承 EventEmitter

大多数时候我们不会直接使用 EventEmitter，而是在对象中继承它。包括 fs、net、 http 在内的，只要是支持事件响应的核心模块都是 EventEmitter 的子类。

为什么要这样做呢？原因有两点：

首先，具有某个实体功能的对象实现事件符合语义， 事件的监听和发射应该是一个对象的方法。

其次 JavaScript 的对象机制是基于原型的，支持 部分多重继承，继承 EventEmitter 不会打乱对象原有的继承关系。

Node.js Buffer(缓冲区)

JavaScript 语言自身只有字符串数据类型，没有二进制数据类型。

但在处理像TCP流或文件流时，必须使用到二进制数据。因此在 Node.js中，定义了一个 Buffer 类，该类用来创建一个专门存放二进制数据的缓存区。

在 Node.js 中，Buffer 类是随 Node 内核一起发布的核心库。Buffer 库为 Node.js 带来了一种存储原始数据的方法，可以让 Node.js 处理二进制数据，每当需要在 Node.js 中处理I/O操作中移动的数据时，就有可能使用 Buffer 库。原始数据存储在 Buffer 类的实例中。一个 Buffer 类似于一个整数数组，但它对应于 V8 堆内存之外的一块原始内存。

创建 Buffer 类

Node Buffer 类可以通过多种方式来创建。

方法 1

创建长度为 10 字节的 Buffer 实例：

```
var buf = new Buffer(10);
```

方法 2

通过给定的数组创建 Buffer 实例：

```
var buf = new Buffer([10, 20, 30, 40, 50]);
```

方法 3

通过一个字符串来创建 Buffer 实例：

```
var buf = new Buffer("www.runoob.com", "utf-8");
```

utf-8 是默认的编码方式，此外它同样支持以下编码："ascii", "utf8", "utf16le", "ucs2", "base64" 和 "hex"。

写入缓冲区

语法

写入 Node 缓冲区的语法如下所示：

```
buf.write(string[, offset][, length][, encoding])
```

参数

参数描述如下：

- **string** - 写入缓冲区的字符串。
- **offset** - 缓冲区开始写入的索引值，默认为 0 。
- **length** - 写入的字节数，默认为 buffer.length
- **encoding** - 使用的编码。默认为 'utf8' 。

返回值

返回实际写入的大小。如果 buffer 空间不足， 则只会写入部分字符串。

实例

```
buf = new Buffer(256);
len = buf.write("www.runoob.com");

console.log("写入字节数 : "+  len);
```

执行以上代码，输出结果为：

```
$node main.js
写入字节数 : 14
```

从缓冲区读取数据

语法

读取 Node 缓冲区数据的语法如下所示：

```
buf.toString([encoding][, start][, end])
```

参数

参数描述如下：

- **encoding** - 使用的编码。默认为 'utf8' 。
- **start** - 指定开始读取的索引位置，默认为 0。
- **end** - 结束位置，默认为缓冲区的末尾。

返回值

解码缓冲区数据并使用指定的编码返回字符串。

实例

```
buf = new Buffer(26);
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97;
}

console.log( buf.toString('ascii'));       // 输出: abcdefghijklmnopqrstuvwxyz
console.log( buf.toString('ascii',0,5));   // 输出: abcde
console.log( buf.toString('utf8',0,5));    // 输出: abcde
console.log( buf.toString(undefined,0,5)); // 使用 'utf8' 编码, 并输出: abcde
```

执行以上代码，输出结果为：

```
$ node main.js
abcdefghijklmnopqrstuvwxyz
abcde
abcde
abcde
```

将 Buffer 转换为 JSON 对象

语法

将 Node Buffer 转换为 JSON 对象的函数语法格式如下：

```
buf.toJSON()
```

返回值

返回 JSON 对象。

实例

```
var buf = new Buffer('www.runoob.com');
var json = buf.toJSON(buf);

console.log(json);
```

执行以上代码，输出结果为：

```
[ 119, 119, 119, 46, 114, 117, 110, 111, 111, 98, 46, 99, 111, 109 ]
```

缓冲区合并

语法

Node 缓冲区合并的语法如下所示：

```
Buffer.concat(list[, totalLength])
```

参数

参数描述如下：

- **list** - 用于合并的 Buffer 对象数组列表。
- **totalLength** - 指定合并后Buffer对象的总长度。

返回值

返回一个多个成员合并的新 Buffer 对象。

实例

```
var buffer1 = new Buffer('菜鸟教程 ');
var buffer2 = new Buffer('www.runoob.com');
var buffer3 = Buffer.concat([buffer1,buffer2]);
console.log("buffer3 内容: " + buffer3.toString());
```

执行以上代码，输出结果为：

```
buffer3 内容: 菜鸟教程 www.runoob.com
```

缓冲区比较

语法

Node Buffer 比较的函数语法如下所示：

```
buf.compare(otherBuffer);
```

参数

参数描述如下：

- **otherBuffer** - 与 **buf** 对象比较的另外一个 Buffer 对象。

返回值

返回一个数字，表示 **buf** 在 **otherBuffer** 之前，之后或相同。

实例

```
var buffer1 = new Buffer('ABC');
var buffer2 = new Buffer('ABCD');
var result = buffer1.compare(buffer2);

if(result < 0) {
   console.log(buffer1 + " 在 " + buffer2 + "之前");
}else if(result == 0){
   console.log(buffer1 + " 与 " + buffer2 + "相同");
}else {
   console.log(buffer1 + " 在 " + buffer2 + "之后");
}
```

执行以上代码，输出结果为：

```
ABC在ABCD之前
```

拷贝缓冲区

语法

Node 缓冲区拷贝语法如下所示：

```
buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])
```

参数

参数描述如下：

- **targetBuffer** - 要拷贝的 Buffer 对象。
- **targetStart** - 数字, 可选, 默认: 0
- **sourceStart** - 数字, 可选, 默认: 0
- **sourceEnd** - 数字, 可选, 默认: buffer.length

返回值

没有返回值。

实例

```
var buffer1 = new Buffer('ABC');
// 拷贝一个缓冲区
var buffer2 = new Buffer(3);
buffer1.copy(buffer2);
console.log("buffer2 content: " + buffer2.toString());
```

执行以上代码，输出结果为：

```
buffer2 content: ABC
```

缓冲区裁剪

Node 缓冲区裁剪语法如下所示：

```
buf.slice([start][, end])
```

参数

参数描述如下：

- **start** - 数字, 可选, 默认: 0
- **end** - 数字, 可选, 默认: buffer.length

返回值

返回一个新的缓冲区，它和旧缓冲区指向同一块内存，但是从索引 start 到 end 的位置剪切。

实例

```
var buffer1 = new Buffer('runoob');
// 剪切缓冲区
var buffer2 = buffer1.slice(0,2);
console.log("buffer2 content: " + buffer2.toString());
```

执行以上代码，输出结果为：

```
buffer2 content: ru
```

缓冲区长度

语法

Node 缓冲区长度计算语法如下所示：

```
buf.length;
```

返回值

返回 Buffer 对象所占据的内存长度。

实例

```
var buffer = new Buffer('www.runoob.com');
//  缓冲区长度
console.log("buffer length: " + buffer.length);
```

执行以上代码，输出结果为：

```
buffer length: 14
```

方法参考手册

以下列出了 Node.js Buffer 模块常用的方法（注意有些方法在旧版本是没有的）：

方法	描述
new Buffer(size)	分配一个新的 size 大小单位为8位字节的 buffer。 注意, size 将会抛出异常 RangeError。
new Buffer(buffer)	拷贝参数 buffer 的数据到 Buffer 实例。

new Buffer(str[, encoding])	分配一个新的 buffer ，其中包含着传入的 str 字符串。 enco
buf.length	返回这个 buffer 的 bytes 数。注意这未必是 buffer 里面内容 分配的内存数，它不会随着这个 buffer 对象内容的改变而改
buf.write(string[, offset][, length][, encoding])	根据参数 offset 偏移量和指定的 encoding 编码方式，将参数 偏移量默认值是 0, encoding 编码方式默认是 utf8。 length 大小。 返回 number 类型，表示写入了多少 8 位字节流。如 个 string，它将只会只写入部分字符串。 length 默认是 buff 出现写入部分字符。
buf.writeUIntLE(value, offset, byteLength[, noAssert])	将value 写入到 buffer 里， 它由offset 和 byteLength 决定， `var b = new Buffer(6); b.writeUIntBE(0x1234567890ab, 0, 6` noAssert 值为 true 时，不再验证 value 和 offset 的有效性。
buf.writeUIntBE(value, offset, byteLength[, noAssert])	将value 写入到 buffer 里， 它由offset 和 byteLength 决定， true 时，不再验证 value 和 offset 的有效性。 默认是 false。
buf.writeIntLE(value, offset, byteLength[, noAssert])	将value 写入到 buffer 里， 它由offset 和 byteLength 决定， true 时，不再验证 value 和 offset 的有效性。 默认是 false。
buf.writeIntBE(value, offset, byteLength[, noAssert])	将value 写入到 buffer 里， 它由offset 和 byteLength 决定， true 时，不再验证 value 和 offset 的有效性。 默认是 false。
buf.readUIntLE(offset, byteLength[, noAssert])	支持读取 48 位以下的数字。noAssert 值为 true 时， offset 度，默认为 false。
buf.readUIntBE(offset, byteLength[, noAssert])	支持读取 48 位以下的数字。noAssert 值为 true 时， offset 度，默认为 false。
buf.readIntLE(offset, byteLength[, noAssert])	支持读取 48 位以下的数字。noAssert 值为 true 时， offset 度，默认为 false。
buf.readIntBE(offset, byteLength[, noAssert])	支持读取 48 位以下的数字。noAssert 值为 true 时， offset 度，默认为 false。
buf.toString([encoding][, start][, end])	根据 encoding 参数（默认是 'utf8'）返回一个解码过的 strin start (默认是 0) 和 end (默认是 buffer.length)作为取值范围。
buf.toJSON()	将 Buffer 实例转换为 JSON 对象。
buf[index]	获取或设置指定的字节。返回值代表一个字节，所以返回值 0xFF 或者十进制0至 255。
buf.equals(otherBuffer)	比较两个缓冲区是否相等，如果是返回 true，否则返回 false
buf.compare(otherBuffer)	比较两个 Buffer 对象，返回一个数字，表示 buf 在 otherBu
buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])	buffer 拷贝，源和目标可以相同。 targetStart 目标开始偏移 是 0。 sourceEnd 源结束位置偏移默认是源的长度 buffer.le
buf.slice([start][, end])	剪切 Buffer 对象，根据 start(默认是 0) 和 end (默认是 buff 负的索引是从 buffer 尾部开始计算的。

buf.readUInt8(offset[, noAssert])	根据指定的偏移量，读取一个有符号 8 位整数。若参数 noA 移量参数。 如果这样 offset 可能会超出buffer 的末尾。默认
buf.readUInt16LE(offset[, noAssert])	根据指定的偏移量，使用特殊的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readUInt16BE(offset[, noAssert])	根据指定的偏移量，使用特殊的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readUInt32LE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readUInt32BE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readInt8(offset[, noAssert])	根据指定的偏移量，读取一个 signed 8 位整数。 若参数 no 偏移量参数。 这意味着 offset 可能会超出 buffer 的末尾。默
buf.readInt16LE(offset[, noAssert])	根据指定的偏移量，使用特殊的 endian 格式读取一个 signe 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.readInt16BE(offset[, noAssert])	根据指定的偏移量，使用特殊的 endian 格式读取一个 signe 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.readInt32LE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readInt32BE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readFloatLE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.readFloatBE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.readDoubleLE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian字节序格式读取一个 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.readDoubleBE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian字节序格式读取一个 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.writeUInt8(value, offset[, noAssert])	根据传入的 offset 偏移量将 value 写入 buffer。注意：value 数。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数 者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。

	否则不要使用。默认是 false。
buf.writeUInt16LE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 个合法的有符号 16 位整数。 若参数 noAssert 为 true 将不 数。 这意味着 value 可能过大，或者 offset 可能会超出buff 除非你对这个参数非常有把握，否则尽量不要使用。默认是
buf.writeUInt16BE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 个合法的有符号 16 位整数。 若参数 noAssert 为 true 将不 数。 这意味着 value 可能过大，或者 offset 可能会超出buff 除非你对这个参数非常有把握，否则尽量不要使用。默认是
buf.writeUInt32LE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 合法的有符号 32 位整数。 若参数 noAssert 为 true 将不会 这意味着value 可能过大，或者offset可能会超出buffer的末 对这个参数非常有把握，否则尽量不要使用。默认是 false。
buf.writeUInt32BE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 合法的有符号 32 位整数。 若参数 noAssert 为 true 将不会 这意味着value 可能过大，或者offset可能会超出buffer的末 对这个参数非常有把握，否则尽量不要使用。默认是 false。
buf.writeInt8(value, offset[, noAssert])	
buf.writeInt16LE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 个合法的 signed 16 位整数。 若参数 noAssert 为 true 将不 数。 这意味着 value 可能过大，或者 offset 可能会超出 buf 弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.writeInt16BE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 个合法的 signed 16 位整数。 若参数 noAssert 为 true 将不 数。 这意味着 value 可能过大，或者 offset 可能会超出 buf 弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.writeInt32LE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 个合法的 signed 32 位整数。 若参数 noAssert 为 true 将不 数。 这意味着 value 可能过大，或者 offset 可能会超出 buf 弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.writeInt32BE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 个合法的 signed 32 位整数。 若参数 noAssert 为 true 将不 数。 这意味着 value 可能过大，或者 offset 可能会超出 buf 弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.writeFloatLE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 个 32 位浮点数类型的值时，结果将是不确定的。 若参数 no 和 offset 偏移量参数。 这意味着 value可能过大，或者 offse 造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽
buf.writeFloatBE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 个 32 位浮点数类型的值时，结果将是不确定的。 若参数 no 和 offset 偏移量参数。 这意味着 value可能过大，或者 offse 造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽
buf.writeDoubleLE(value,	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 个有效的 64 位double 类型的值。 若参数 noAssert 为 true

offset[, noAssert])	参数。 这意味着 value 可能过大，或者 offset 可能会超出 b 弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.writeDoubleBE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 个有效的 64 位double 类型的值。 若参数 noAssert 为 true 参数。 这意味着 value 可能过大，或者 offset 可能会超出 b 弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.fill(value[, offset][, end])	使用指定的 value 来填充这个 buffer。如果没有指定 offset (buffer.length)， 将会填充整个buffer。

Node.js Stream(流)

Stream 是一个抽象接口，Node 中有很多对象实现了这个接口。例如，对http 服务器发起请求的request 对象就是一个 Stream，还有stdout（标准输出）。

Node.js，Stream 有四种流类型：

- **Readable** - 可读操作。
- **Writable** - 可写操作。
- **Duplex** - 可读可写操作.
- **Transform** - 操作被写入数据，然后读出结果。

所有的 Stream 对象都是 EventEmitter 的实例。常用的事件有：

- **data** - 当有数据可读时触发。
- **end** - 没有更多的数据可读时触发。
- **error** - 在接收和写入过程中发生错误时触发。
- **finish** - 所有数据已被写入到底层系统时触发。

本教程会为大家介绍常用的流操作。

从流中读取数据

创建 input.txt 文件，内容如下：

```
菜鸟教程官网地址：www.runoob.com
```

创建 main.js 文件, 代码如下：

```
var fs = require("fs");
var data = '';

// 创建可读流
var readerStream = fs.createReadStream('input.txt');

// 设置编码为 utf8。
readerStream.setEncoding('UTF8');

// 处理流事件 --> data, end, and error
readerStream.on('data', function(chunk) {
   data += chunk;
});

readerStream.on('end',function(){
   console.log(data);
});

readerStream.on('error', function(err){
   console.log(err.stack);
});

console.log("程序执行完毕");
```

以上代码执行结果如下：

```
程序执行完毕
菜鸟教程官网地址：www.runoob.com
```

写入流

创建 main.js 文件, 代码如下：

```
var fs = require("fs");
var data = '菜鸟教程官网地址：www.runoob.com';

// 创建一个可以写入的流，写入到文件 output.txt 中
var writerStream = fs.createWriteStream('output.txt');

// 使用 utf8 编码写入数据
writerStream.write(data,'UTF8');

// 标记文件末尾
writerStream.end();

// 处理流事件 --> data, end, and error
writerStream.on('finish', function() {
    console.log("写入完成。");
});

writerStream.on('error', function(err){
   console.log(err.stack);
});

console.log("程序执行完毕");
```

以上程序会将 data 变量的数据写入到 output.txt 文件中。代码执行结果如下：

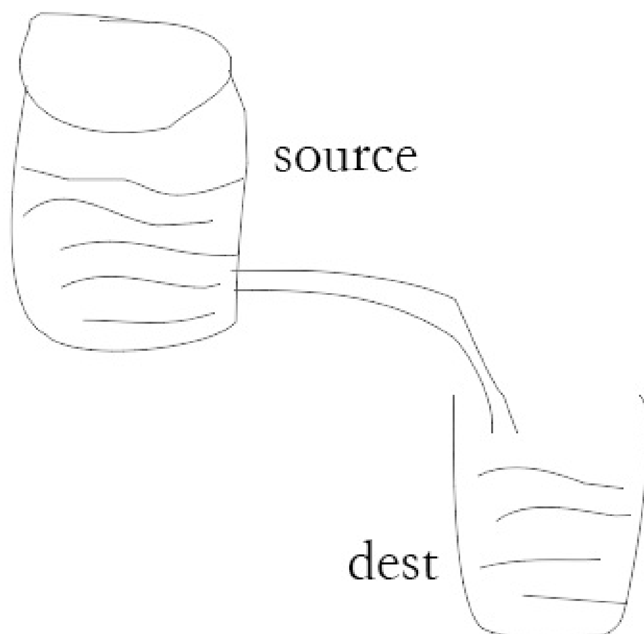
```
$ node main.js
程序执行完毕
写入完成。
```

查看 output.txt 文件的内容：

```
$ cat output.txt
菜鸟教程官网地址：www.runoob.com
```

管道流

管道提供了一个输出流到输入流的机制。通常我们用于从一个流中获取数据并将数据传递到另外一个流中。



如上面的图片所示，我们把文件比作装水的桶，而水就是文件里的内容，我们用一根管子(pipe)连接两个桶使得水从一个桶流入另一个桶，这样就慢慢的实现了大文件的复制过程。

以下实例我们通过读取一个文件内容并将内容写入到另外一个文件中。

设置 input.txt 文件内容如下：

```
菜鸟教程官网地址：www.runoob.com
管道流操作实例
```

创建 main.js 文件, 代码如下：

```
var fs = require("fs");

// 创建一个可读流
var readerStream = fs.createReadStream('input.txt');

// 创建一个可写流
var writerStream = fs.createWriteStream('output.txt');

// 管道读写操作
// 读取 input.txt 文件内容，并将内容写入到 output.txt 文件中
readerStream.pipe(writerStream);

console.log("程序执行完毕");
```

代码执行结果如下：

```
$ node main.js
程序执行完毕
```

查看 output.txt 文件的内容：

```
$ cat output.txt
菜鸟教程官网地址：www.runoob.com
管道流操作实例
```

链式流

链式是通过连接输出流到另外一个流并创建多个对个流操作链的机制。链式流一般用于管道操作。

接下来我们就是用管道和链式来压缩和解压文件。

创建 compress.js 文件, 代码如下：

```
var fs = require("fs");
var zlib = require('zlib');

// 压缩 input.txt 文件为 input.txt.gz
fs.createReadStream('input.txt')
  .pipe(zlib.createGzip())
  .pipe(fs.createWriteStream('input.txt.gz'));

console.log("文件压缩完成。");
```

代码执行结果如下：

```
$ node compress.js
文件压缩完成。
```

执行完以上操作后，我们可以看到当前目录下生成了 input.txt 的压缩文件 input.txt.gz。

接下来，让我们来解压该文件，创建 decompress.js 文件，代码如下：

```
var fs = require("fs");
var zlib = require('zlib');

// 解压 input.txt.gz 文件为 input.txt
fs.createReadStream('input.txt.gz')
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream('input.txt'));

console.log("文件解压完成。");
```

代码执行结果如下：

```
$ node decompress.js
文件解压完成。
```

Node.js模块系统

为了让Node.js的文件可以相互调用，Node.js提供了一个简单的模块系统。

模块是Node.js 应用程序的基本组成部分，文件和模块是一一对应的。换言之，一个 Node.js 文件就是一个模块，这个文件可能是JavaScript 代码、JSON 或者编译过的C/C++ 扩展。

创建模块

在 Node.js 中，创建一个模块非常简单，如下我们创建一个 'main.js' 文件，代码如下:

```
var hello = require('./hello');
hello.world();
```

以上实例中，代码 require('./hello') 引入了当前目录下的hello.js文件（./ 为当前目录，node.js 默认后缀为js）。

Node.js 提供了exports 和 require 两个对象，其中 exports 是模块公开的接口，require 用于从外部获取一个模块的接口，即所获取模块的 exports 对象。

接下来我们就来创建hello.js文件，代码如下：

```
exports.world = function() {
  console.log('Hello World');
}
```

在以上示例中，hello.js 通过 exports 对象把 world 作为模块的访 问接口，在 main.js 中通过 require('./hello') 加载这个模块，然后就可以直接访 问main.js 中 exports 对象的成员函数了。

有时候我们只是想把一个对象封装到模块中，格式如下：

```
module.exports = function() {
  // ...
}
```

例如:

```
//hello.js
function Hello() {
    varname;
    this.setName = function(thyName) {
        name = thyName;
    };
    this.sayHello = function() {
        console.log('Hello ' + name);
    };
};
module.exports = Hello;
```

这样就可以直接获得这个对象了：

```
//main.js
var Hello = require('./hello');
hello = new Hello();
hello.setName('BYVoid');
hello.sayHello();
```

模块接口的唯一变化是使用 module.exports = Hello 代替了exports.world = function(){}。 在外部引用该模块时，其接口对象就是要输出的 Hello 对象本身，而不是原先的 exports。

服务端的模块放在哪里

也许你已经注意到，我们已经在代码中使用了模块了。像这样：

```
var http = require("http");

...

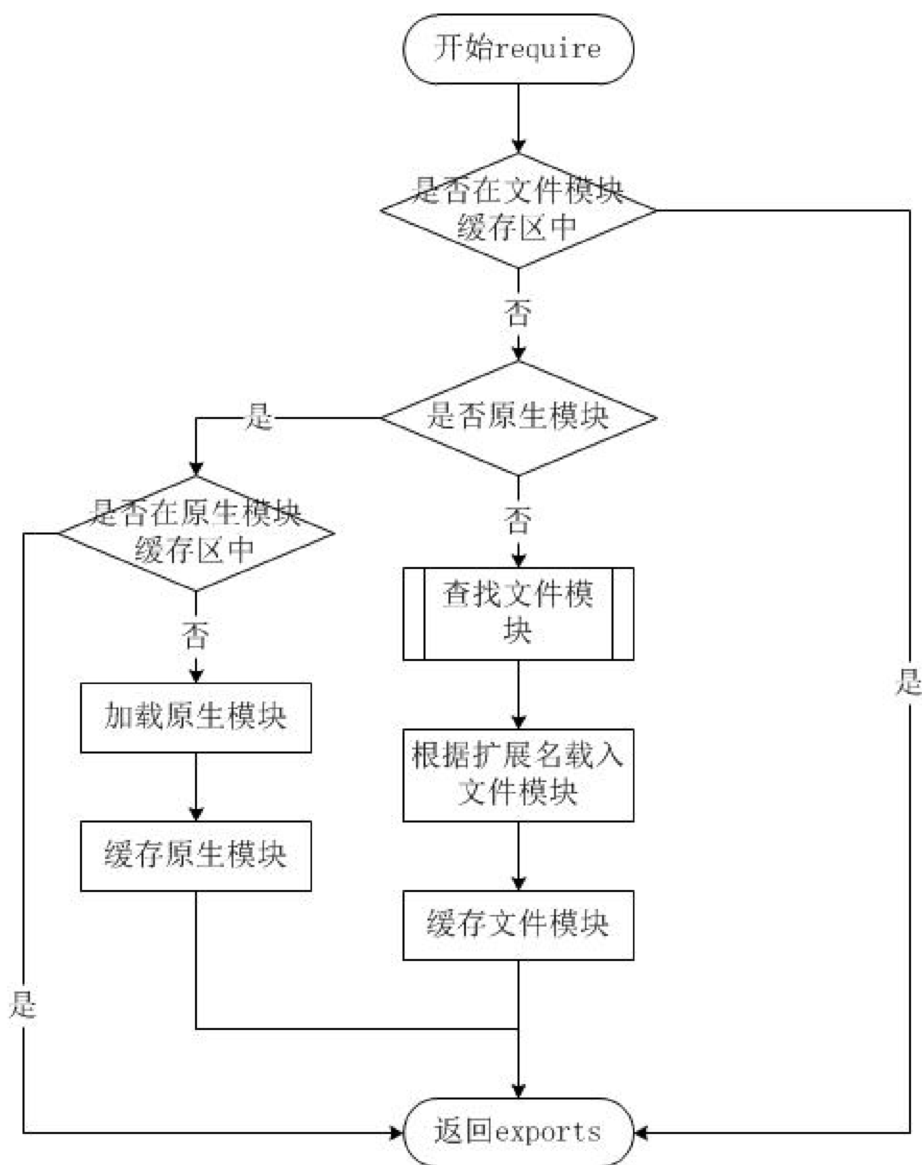
http.createServer(...);
```

Node.js中自带了一个叫做"http"的模块，我们在我们的代码中请求它并把返回值赋给一个本地变量。

这把我们的本地变量变成了一个拥有所有 http 模块所提供的公共方法的对象。

Node.js 的 require方法中的文件查找策略如下：

由于Node.js中存在4类模块（原生模块和3种文件模块），尽管require方法极其简单，但是内部的加载却是十分复杂的，其加载优先级也各自不同。如下图所示：



从文件模块缓存中加载

尽管原生模块与文件模块的优先级不同，但是都不会优先于从文件模块的缓存中加载已经存在的模块。

从原生模块加载

原生模块的优先级仅次于文件模块缓存的优先级。require方法在解析文件名之后，优先检查模块是否在原生模块列表中。以http模块为例，尽管在目录下存在一个http/http.js/http.node/http.json文件，require("http")都不会从这些文件中加载，而是从原生模块中加载。

原生模块也有一个缓存区，同样也是优先从缓存区加载。如果缓存区没有被加载过，则调用原生模块的加载方式进行加载和执行。

从文件加载

当文件模块缓存中不存在，而且不是原生模块的时候，Node.js会解析require方法传入的参数，并从文件系统中加载实际的文件，加载过程中的包装和编译细节在前一节中已经介绍过，这里我们将详细描述查找文件模块的过程，其中，也有一些细节值得知晓。

require方法接受以下几种参数的传递：

- http、fs、path等，原生模块。
- ./mod或../mod，相对路径的文件模块。
- /pathtomodule/mod，绝对路径的文件模块。
- mod，非原生模块的文件模块。

Node.js 函数

在JavaScript中，一个函数可以作为另一个函数接收一个参数。我们可以先定义一个函数，然后传递，也可以在传递参数的地方直接定义函数。

Node.js中函数的使用与Javascript类似，举例来说，你可以这样做：

```
function say(word) {
  console.log(word);
}

function execute(someFunction, value) {
  someFunction(value);
}

execute(say, "Hello");
```

以上代码中，我们把 say 函数作为execute函数的第一个变量进行了传递。这里返回的不是 say 的返回值，而是 say 本身！

这样一来， say 就变成了execute 中的本地变量 someFunction ， execute可以通过调用 someFunction() （带括号的形式）来使用 say 函数。

当然，因为 say 有一个变量， execute 在调用 someFunction 时可以传递这样一个变量。

匿名函数

我们可以把一个函数作为变量传递。但是我们不一定要绕这个"先定义，再传递"的圈子，我们可以直接在另一个函数的括号中定义和传递这个函数：

```
function execute(someFunction, value) {
  someFunction(value);
}

execute(function(word){ console.log(word) }, "Hello");
```

我们在 execute 接受第一个参数的地方直接定义了我们准备传递给 execute 的函数。

用这种方式，我们甚至不用给这个函数起名字，这也是为什么它被叫做匿名函数 。

函数传递是如何让HTTP服务器工作的

带着这些知识，我们再来看看我们简约而不简单的HTTP服务器：

```
var http = require("http");

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

现在它看上去应该清晰了很多：我们向 createServer 函数传递了一个匿名函数。

用这样的代码也可以达到同样的目的：

```
var http = require("http");

function onRequest(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}

http.createServer(onRequest).listen(8888);
```

Node.js 路由

我们要为路由提供请求的URL和其他需要的GET及POST参数，随后路由需要根据这些数据来执行相应的代码。

因此，我们需要查看HTTP请求，从中提取出请求的URL以及GET/POST参数。这一功能应当属于路由还是服务器（甚至作为一个模块自身的功能）确实值得探讨，但这里暂定其为我们的HTTP服务器的功能。

我们需要的所有数据都会包含在request对象中，该对象作为onRequest()回调函数的第一个参数传递。但是为了解析这些数据，我们需要额外的Node.JS模块，它们分别是url和querystring模块。

```
                     url.parse(string).query
                                           |
             url.parse(string).pathname      |
                         |                   |
                         |                   |
                       ------ -------------------
http://localhost:8888/start?foo=bar&hello=world
                                    ---       -----
                                     |          |
                                     |          |
                  querystring(string)["foo"]    |
                                                |
                             querystring(string)["hello"]
```

当然我们也可以用querystring模块来解析POST请求体中的参数，稍后会有演示。

现在我们来给onRequest()函数加上一些逻辑，用来找出浏览器请求的URL路径：

```
var http = require("http");
var url = require("url");

function start() {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

好了，我们的应用现在可以通过请求的URL路径来区别不同请求了--这使我们得以使用路由（还未完成）来将请求以URL路径为基准映射到处理程序上。

在我们所要构建的应用中，这意味着来自/start和/upload的请求可以使用不同的代码来处理。稍后我们将看到这些内容是如何整合到一起的。

现在我们可以来编写路由了，建立一个名为router.js的文件，添加以下内容：

```
function route(pathname) {
  console.log("About to route a request for " + pathname);
}

exports.route = route;
```

如你所见，这段代码什么也没干，不过对于现在来说这是应该的。在添加更多的逻辑以前，我们先来看看如何把路由和服务器整合起来。

我们的服务器应当知道路由的存在并加以有效利用。我们当然可以通过硬编码的方式将这一依赖项绑定到服务器上，但是其它语言的编程经验告诉我们这会是一件非常痛苦的事，因此我们将使用依赖注入的方式较松散地添加路由模块。

首先，我们来扩展一下服务器的start()函数，以便将路由函数作为参数传递过去：

```
var http = require("http");
var url = require("url");

function start(route) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

    route(pathname);

    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

同时，我们会相应扩展index.js，使得路由函数可以被注入到服务器中：

```
var server = require("./server");
var router = require("./router");

server.start(router.route);
```

在这里，我们传递的函数依旧什么也没做。

如果现在启动应用（node index.js，始终记得这个命令行），随后请求一个URL，你将会看到应用输出相应的信息，这表明我们的HTTP服务器已经在使用路由模块了，并会将请求的路径传递给路由：

```
bash$ node index.js
Request for /foo received.
About to route a request for /foo
```

以上输出已经去掉了比较烦人的/favicon.ico请求相关的部分。

Node.js 全局对象

JavaScript 中有一个特殊的对象，称为全局对象（Global Object），它及其所有属性都可 以在程序的任何地方访问，即全局变量。

在浏览器JavaScript 中，通常window 是全局对象， 而Node.js 中的全局对象是 global，所有全局变量（除了 global 本身以外）都是 global 对象的属性。

我们在Node.js 中能够直接访问到对象通常都是 global 的属性，如 console、process 等，下面逐一介绍。

全局对象与全局变量

global 最根本的作用是作为全局变量的宿主。按照ECMAScript 的定义，满足以下条 件的变量是全局变量：

- 在最外层定义的变量；
- 全局对象的属性；
- 隐式定义的变量（未定义直接赋值的变量）。

当你定义一个全局变量时，这个变量同时也会成为全局对象的属性，反之亦然。需要注 意的是，在Node.js 中你不可能在最外层定义变量，因为所有用户代码都是属于当前模块的， 而模块本身不是最外层上下文。

注意： 永远使用var 定义变量以避免引入全局变量，因为全局变量会污染 命名空间，提高代码的耦合风险。

process

process 是一个全局变量，即 global 对象的属性。

它用于描述当前Node.js 进程状态 的对象，提供了一个与操作系统的简单接口。通常在你写本地命令行程序的时候，少不了要 和它打交道。下面将会介绍process 对象的一些最常用的成员方法。

process.argv是命令行参数数组，第一个元素是 node，第二个元素是脚本文件名， 从第三个元素开始每个元素是一个运行参数。

```
console.log(process.argv);
```

将以上代码存储为argv.js，通过以下命令运行：

```
$ node argv.js 1991 name=byvoid --v "Carbo Kuo"
[ 'node',
'/home/byvoid/argv.js',
'1991',
'name=byvoid',
'--v',
'Carbo Kuo' ]
```

- **process.stdout**是标准输出流，通常我们使用的 console.log() 向标准输出打印 字符，而 process.stdout.write() 函数提供了更底层的接口。
- **process.stdin**是标准输入流，初始时它是被暂停的，要想从标准输入读取数据， 你必须恢复流，并手动编写流的事件响应函数。

```
process.stdin.resume();
process.stdin.on('data', function(data) {
process.stdout.write('read from console: ' + data.toString());
});
```

- **process.nextTick(callback)**的功能是为事件循环设置一项任务，Node.js 会在 下次事件循环调响应时调用 callback。

初学者很可能不理解这个函数的作用，有什么任务不能在当下执行完，需要交给下次事 件循环响应来做呢？

我们讨论过，Node.js 适合I/O 密集型的应用，而不是计算密集型的应用， 因为一个Node.js 进程只有一个线程，因此在任何时刻都只有一个事件在执行。

如果这个事 件占用大量的CPU 时间，执行事件循环中的下一个事件就需要等待很久，因此 Node.js 的一 个编程原则就是尽量缩短每个事件的执行时间。process.nextTick() 提供了一个这样的 工具，可以把复杂的工作拆散， 变成一个个较小的事件。

```
functiondoSomething(args, callback) {
    somethingComplicated(args);
    callback();
}
doSomething(functiononEnd() {
    compute();
});
```

我们假设compute() 和somethingComplicated() 是两个较为耗时的函数，以上 的程序在调用 doSomething() 时会先执行somethingComplicated()，然后立即调用 回调函数，在 onEnd() 中又会执行 compute()。下面用process.nextTick() 改写上 面的程序：

```
functiondoSomething(args, callback) {
    somethingComplicated(args);
    process.nextTick(callback);
}
doSomething(functiononEnd() {
    compute();
});
```

改写后的程序会把上面耗时的操作拆分为两个事件，减少每个事件的执行时间，提高事 件响应速度。

注意： 不要使用setTimeout(fn,0)代替process.nextTick(callback)， 前者比后者效率要低得多。

我们探讨了process对象常用的几个成员，除此之外process还展示了process.platform、process.pid、process.execPath、process.memoryUsage() 等方法，以及POSIX 进程信号响应机制。有兴趣的读者可以访问http://nodejs.org/api/process.html 了解详细 内容。

console

console 用于提供控制台标准输出，它是由Internet Explorer 的JScript 引擎提供的调试 工具，后来逐渐成为浏览器的事实标准。

Node.js 沿用了这个标准，提供与习惯行为一致的 console 对象，用于向标准输出流（stdout）或标准错误流（stderr）输出字符。 ? console.log()：向标准输出流打印字符并以换行符结束。

console.log 接受若干 个参数，如果只有一个参数，则输出这个参数的字符串形式。如果有多个参数，则 以类似于C 语言 printf() 命令的格式输出。

第一个参数是一个字符串，如果没有 参数，只打印一个换行。

```
console.log('Hello world');
console.log('byvoid%diovyb');
console.log('byvoid%diovyb', 1991);
```

运行结果为：

```
Hello world
byvoid%diovyb
byvoid1991iovyb
```

- console.error()：与console.log() 用法相同，只是向标准错误流输出。
- console.trace()：向标准错误流输出当前的调用栈。

```
console.trace();
```

运行结果为：

```
Trace:
at Object.<anonymous> (/home/byvoid/consoletrace.js:1:71)
at Module._compile (module.js:441:26)
at Object..js (module.js:459:10)
at Module.load (module.js:348:31)
at Function._load (module.js:308:12)
at Array.0 (module.js:479:10)
at EventEmitter._tickCallback (node.js:192:40)
```

Node.js 常用工具 util

util 是一个Node.js 核心模块，提供常用函数的集合，用于弥补核心JavaScript 的功能 过于精简的不足。

util.inherits

util.inherits(constructor, superConstructor)是一个实现对象间原型继承 的函数。

JavaScript 的面向对象特性是基于原型的，与常见的基于类的不同。JavaScript 没有 提供对象继承的语言级别特性，而是通过原型复制来实现的。

在这里我们只介绍util.inherits 的用法，示例如下：

```
varutil = require('util');
functionBase() {
    this.name = 'base';
    this.base = 1991;
    this.sayHello = function() {
    console.log('Hello ' + this.name);
    };
}
Base.prototype.showName = function() {
    console.log(this.name);
};
functionSub() {
    this.name = 'sub';
}
util.inherits(Sub, Base);
varobjBase = newBase();
objBase.showName();
objBase.sayHello();
console.log(objBase);
varobjSub = newSub();
objSub.showName();
//objSub.sayHello();
console.log(objSub);
```

我们定义了一个基础对象Base 和一个继承自Base 的Sub，Base 有三个在构造函数 内定义的属性和一个原型中定义的函数，通过util.inherits 实现继承。运行结果如下：

```
base
Hello base
{ name: 'base', base: 1991, sayHello: [Function] }
sub
{ name: 'sub' }
```

注意：Sub 仅仅继承了Base 在原型中定义的函数，而构造函数内部创造的 base 属 性和 sayHello 函数都没有被 Sub 继承。

同时，在原型中定义的属性不会被console.log 作 为对象的属性输出。如果我们去掉 objSub.sayHello(); 这行的注释，将会看到：

```
node.js:201
throw e; // process.nextTick error, or 'error' event on first tick
^
TypeError: Object #<Sub> has no method 'sayHello'
at Object.<anonymous> (/home/byvoid/utilinherits.js:29:8)
at Module._compile (module.js:441:26)
at Object..js (module.js:459:10)
at Module.load (module.js:348:31)
at Function._load (module.js:308:12)
at Array.0 (module.js:479:10)
at EventEmitter._tickCallback (node.js:192:40)
```

util.inspect

util.inspect(object,[showHidden],[depth],[colors])是一个将任意对象转换 为字符串的方法，通常用于调试和错误输出。它至少接受一个参数 object，即要转换的对象。

showHidden 是一个可选参数，如果值为 true，将会输出更多隐藏信息。

depth 表示最大递归的层数，如果对象很复杂，你可以指定层数以控制输出信息的多 少。如果不指定depth，默认会递归2层，指定为 null 表示将不限递归层数完整遍历对象。 如果color 值为 true，输出格式将会以ANSI 颜色编码，通常用于在终端显示更漂亮 的效果。

特别要指出的是，util.inspect 并不会简单地直接把对象转换为字符串，即使该对 象定义了 toString 方法也不会调用。

```
varutil = require('util');
functionPerson() {
    this.name = 'byvoid';
    this.toString = function() {
    return this.name;
    };
}
varobj = newPerson();
console.log(util.inspect(obj));
console.log(util.inspect(obj, true));
```

运行结果是：

```
{ name: 'byvoid', toString: [Function] }
{ toString:
{ [Function]
[prototype]: { [constructor]: [Circular] },
[caller]: null,
[length]: 0,
[name]: '',
[arguments]: null },
name: 'byvoid' }
```

util.isArray(object)

如果给定的参数 "object" 是一个数组返回true，否则返回false。

```
var util = require('util');

util.isArray([])
  // true
util.isArray(new Array)
  // true
util.isArray({})
  // false
```

util.isRegExp(object)

如果给定的参数 "object" 是一个正则表达式返回true，否则返回false。

```
var util = require('util');

util.isRegExp(/some regexp/)
  // true
util.isRegExp(new RegExp('another regexp'))
  // true
util.isRegExp({})
  // false
```

util.isDate(object)

如果给定的参数 "object" 是一个日期返回true，否则返回false。

```
var util = require('util');

util.isDate(new Date())
  // true
util.isDate(Date())
  // false (without 'new' returns a String)
util.isDate({})
  // false
```

util.isError(object)

如果给定的参数 "object" 是一个错误对象返回true，否则返回false。

```
var util = require('util');

util.isError(new Error())
  // true
util.isError(new TypeError())
  // true
util.isError({ name: 'Error', message: 'an error occurred' })
  // false
```

更多详情可以访问 http://nodejs.org/api/util.html 了解详细内容。

Node.js 文件系统

Node.js 提供一组类似 UNIX（POSIX）标准的文件操作API。 Node 导入文件系统模块(fs)语法如下所示：

```
var fs = require("fs")
```

异步和同步

Node.js 文件系统（fs 模块）模块中的方法均有异步和同步版本，例如读取文件内容的函数有异步的 fs.readFile() 和同步的 fs.readFileSync()。

异步的方法函数最后一个参数为回调函数，回调函数的第一个参数包含了错误信息(error)。

建议大家是用异步方法，比起同步，异步方法性能更高，速度更快，而且没有阻塞。

实例

创建 input.txt 文件，内容如下：

```
菜鸟教程官网地址：www.runoob.com
文件读取实例
```

创建 file.js 文件, 代码如下：

```
var fs = require("fs");

// 异步读取
fs.readFile('input.txt', function (err, data) {
   if (err) {
       return console.error(err);
   }
   console.log("异步读取: " + data.toString());
});

// 同步读取
var data = fs.readFileSync('input.txt');
console.log("同步读取: " + data.toString());

console.log("程序执行完毕。");
```

以上代码执行结果如下：

```
$ node file.js
同步读取: 菜鸟教程官网地址：www.runoob.com
文件读取实例

程序执行完毕。
异步读取: 菜鸟教程官网地址：www.runoob.com
文件读取实例
```

接下来，让我们来具体了解下 Node.js 文件系统的方法。

打开文件

语法

以下为在异步模式下打开文件的语法格式：

```
fs.open(path, flags[, mode], callback)
```

参数

参数使用说明如下：

- **path** - 文件的路径。
- **flags** - 文件打开的行为。具体值详见下文。
- **mode** - 设置文件模式(权限)，文件创建默认权限为 0666(可读，可写)。
- **callback** - 回调函数，带有两个参数如：callback(err, fd)。

flags 参数可以是以下值：

Flag	描述
r	以读取模式打开文件。如果文件不存在抛出异常。
r+	以读写模式打开文件。如果文件不存在抛出异常。
rs	以同步的方式读取文件。
rs+	以同步的方式读取和写入文件。
w	以写入模式打开文件，如果文件不存在则创建。
wx	类似 'w'，但是如果文件路径不存在，则文件写入失败。
w+	以读写模式打开文件，如果文件不存在则创建。
wx+	类似 'w+'， 但是如果文件路径不存在，则文件读写失败。
a	以追加模式打开文件，如果文件不存在则创建。
ax	类似 'a'， 但是如果文件路径不存在，则文件追加失败。
a+	以读取追加模式打开文件，如果文件不存在则创建。
ax+	类似 'a+'， 但是如果文件路径不存在，则文件读取追加失败。

实例

接下来我们创建 file.js 文件，并打开 input.txt 文件进行读写，代码如下所示：

```
var fs = require("fs");

// 异步打开文件
console.log("准备打开文件！");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
       return console.error(err);
   }
  console.log("文件打开成功！");
});
```

以上代码执行结果如下：

```
$ node file.js
准备打开文件！
文件打开成功！
```

获取文件信息

语法

以下为通过异步模式获取文件信息的语法格式：

```
fs.stat(path, callback)
```

参数

参数使用说明如下：

- **path** - 文件路径。
- **callback** - 回调函数，带有两个参数如：(err, stats), **stats** 是 fs.Stats 对象。

fs.stat(path)执行后，会将stats类的实例返回给其回调函数。可以通过stats类中的提供方法判断文件的相关属性。例如判断是否为文件：

```
var fs = require('fs');

fs.stat('/Users/liuht/code/itbilu/demo/fs.js', function (err, stats) {
    console.log(stats.isFile());         //true
})
```

stats类中的方法有：

方法	描述
stats.isFile()	如果是文件返回 true，否则返回 false。
stats.isDirectory()	如果是目录返回 true，否则返回 false。
stats.isBlockDevice()	如果是块设备返回 true，否则返回 false。
stats.isCharacterDevice()	如果是字符设备返回 true，否则返回 false。
stats.isSymbolicLink()	如果是软链接返回 true，否则返回 false。
stats.isFIFO()	如果是FIFO，返回true，否则返回 false。FIFO是UNIX中的一种特殊类型的命令管道。
stats.isSocket()	如果是 Socket 返回 true，否则返回 false。

实例

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log("准备打开文件！");
fs.stat('input.txt', function (err, stats) {
   if (err) {
       return console.error(err);
   }
   console.log(stats);
   console.log("读取文件信息成功！");

   // 检测文件类型
   console.log("是否为文件(isFile) ? " + stats.isFile());
   console.log("是否为目录(isDirectory) ? " + stats.isDirectory());
});
```

以上代码执行结果如下：

```
$ node file.js
准备打开文件！
{ dev: 16777220,
  mode: 33188,
  nlink: 1,
  uid: 501,
  gid: 20,
  rdev: 0,
  blksize: 4096,
  ino: 40333161,
  size: 61,
  blocks: 8,
  atime: Mon Sep 07 2015 17:43:55 GMT+0800 (CST),
  mtime: Mon Sep 07 2015 17:22:35 GMT+0800 (CST),
  ctime: Mon Sep 07 2015 17:22:35 GMT+0800 (CST) }
读取文件信息成功！
是否为文件(isFile) ? true
是否为目录(isDirectory) ? false
```

写入文件

语法

以下为异步模式下写入文件的语法格式：

```
fs.writeFile(filename, data[, options], callback)
```

如果文件存在，该方法写入的内容会覆盖旧的文件内容。

参数

参数使用说明如下：

- **path** - 文件路径。
- **data** - 要写入文件的数据，可以是 String(字符串) 或 Buffer(流) 对象。

- **options** - 该参数是一个对象，包含 {encoding, mode, flag}。默认编码为 utf8, 模式为 0666 ， flag 为 'w'
- **callback** - 回调函数，回调函数只包含错误信息参数(err)，在写入失败时返回。

实例

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log("准备写入文件");
fs.writeFile('input.txt', '我是通过写入的文件内容！',  function(err) {
   if (err) {
       return console.error(err);
   }
   console.log("数据写入成功！");
   console.log("--------我是分割线-------------")
   console.log("读取写入的数据！");
   fs.readFile('input.txt', function (err, data) {
      if (err) {
         return console.error(err);
      }
      console.log("异步读取文件数据: " + data.toString());
   });
});
```

以上代码执行结果如下：

```
$ node file.js
准备写入文件
数据写入成功！
--------我是分割线-------------
读取写入的数据！
异步读取文件数据: 我是通过写入的文件内容
```

读取文件

语法

以下为异步模式下读取文件的语法格式：

```
fs.read(fd, buffer, offset, length, position, callback)
```

该方法使用了文件描述符来读取文件。

参数

参数使用说明如下：

- **fd** - 通过 fs.open() 方法返回的文件描述符。
- **buffer** - 数据写入的缓冲区。
- **offset** - 缓冲区写入的写入偏移量。
- **length** - 要从文件中读取的字节数。
- **position** - 文件读取的起始位置，如果 position 的值为 null，则会从当前文件指针的位置读取。
- **callback** - 回调函数，有三个参数err, bytesRead, buffer，err 为错误信息， bytesRead 表示读取的字节数，buffer 为缓冲区对象。

实例

input.txt 文件内容为：

```
菜鸟教程官网地址：www.runoob.com
```

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");
var buf = new Buffer(1024);

console.log("准备打开已存在的文件！");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
       return console.error(err);
   }
   console.log("文件打开成功！");
   console.log("准备读取文件：");
   fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
      if (err){
         console.log(err);
      }
      console.log(bytes + "  字节被读取");

      // 仅输出读取的字节
      if(bytes > 0){
         console.log(buf.slice(0, bytes).toString());
      }
   });
});
```

以上代码执行结果如下：

```
$ node file.js
准备打开已存在的文件！
文件打开成功！
准备读取文件：
42  字节被读取
菜鸟教程官网地址：www.runoob.com
```

关闭文件

语法

以下为异步模式下关闭文件的语法格式：

```
fs.close(fd, callback)
```

该方法使用了文件描述符来读取文件。

参数

参数使用说明如下：

- **fd** - 通过 fs.open() 方法返回的文件描述符。
- **callback** - 回调函数，没有参数。

实例

input.txt 文件内容为：

```
菜鸟教程官网地址：www.runoob.com
```

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");
var buf = new Buffer(1024);

console.log("准备打开文件！");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
       return console.error(err);
   }
   console.log("文件打开成功！");
   console.log("准备读取文件！");
   fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
      if (err){
         console.log(err);
      }

      // 仅输出读取的字节
      if(bytes > 0){
         console.log(buf.slice(0, bytes).toString());
      }

      // 关闭文件
      fs.close(fd, function(err){
         if (err){
            console.log(err);
         }
         console.log("文件关闭成功");
      });
   });
});
```

以上代码执行结果如下：

```
$ node file.js
准备打开文件！
文件打开成功！
准备读取文件！
菜鸟教程官网地址：www.runoob.com
文件关闭成功
```

截取文件

语法

以下为异步模式下截取文件的语法格式：

```
fs.ftruncate(fd, len, callback)
```

该方法使用了文件描述符来读取文件。

参数

参数使用说明如下：

- **fd** - 通过 fs.open() 方法返回的文件描述符。

- **len** - 文件内容截取的长度。
- **callback** - 回调函数，没有参数。

实例

input.txt 文件内容为：

```
site:www.runoob.com
```

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");
var buf = new Buffer(1024);

console.log("准备打开文件！");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
       return console.error(err);
   }
   console.log("文件打开成功！");
   console.log("截取10字节后的文件内容。");

   // 截取文件
   fs.ftruncate(fd, 10, function(err){
      if (err){
         console.log(err);
      }
      console.log("文件截取成功。");
      console.log("读取相同的文件");
      fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
         if (err){
            console.log(err);
         }

         // 仅输出读取的字节
         if(bytes > 0){
            console.log(buf.slice(0, bytes).toString());
         }

         // 关闭文件
         fs.close(fd, function(err){
            if (err){
               console.log(err);
            }
            console.log("文件关闭成功！");
         });
      });
   });
});
```

以上代码执行结果如下：

```
$ node file.js
准备打开文件！
文件打开成功！
截取10字节后的文件内容。
文件截取成功。
读取相同的文件
site:www.r
文件关闭成功
```

删除文件

语法

以下为删除文件的语法格式：

```
fs.unlink(path, callback)
```

参数

参数使用说明如下：

- **path** - 文件路径。
- **callback** - 回调函数，没有参数。

实例

input.txt 文件内容为：

```
site:www.runoob.com
```

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log("准备删除文件！");
fs.unlink('input.txt', function(err) {
   if (err) {
       return console.error(err);
   }
   console.log("文件删除成功！");
});
```

以上代码执行结果如下：

```
$ node file.js
准备删除文件！
文件删除成功！
```

再去查看 input.txt 文件，发现已经不存在了。

创建目录

语法

以下为创建目录的语法格式：

```
fs.mkdir(path[, mode], callback)
```

参数

参数使用说明如下：

- **path** - 文件路径。
- **mode** - 设置目录权限，默认为 0777。
- **callback** - 回调函数，没有参数。

实例

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log(创建目录 /tmp/test");
fs.mkdir('/tmp/test',function(err){
   if (err) {
       return console.error(err);
   }
   console.log("目录创建成功。");
});
```

以上代码执行结果如下：

```
$ node file.js
创建目录 /tmp/test
目录创建成功。
```

读取目录

语法

以下为读取目录的语法格式：

```
fs.readdir(path, callback)
```

参数

参数使用说明如下：

- **path** - 文件路径。
- **callback** - 回调函数，回调函数带有两个参数err, files，err 为错误信息，files 为 目录下的文件数组列表。

实例

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log("查看 /tmp 目录");
fs.readdir("/tmp/",function(err, files){
   if (err) {
       return console.error(err);
   }
   files.forEach( function (file){
       console.log( file );
   });
});
```

以上代码执行结果如下：

```
$ node file.js
查看 /tmp 目录
input.out
output.out
test
test.txt
```

删除目录

语法

以下为删除目录的语法格式：

```
fs.rmdir(path, callback)
```

参数

参数使用说明如下：

- **path** - 文件路径。
- **callback** - 回调函数，没有参数。

实例

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log("准备删除目录 /tmp/test");
fs.rmdir("/tmp/test",function(err){
   if (err) {
       return console.error(err);
   }
   console.log("读取 /tmp 目录");
   fs.readdir("/tmp/",function(err, files){
      if (err) {
          return console.error(err);
      }
      files.forEach( function (file){
          console.log( file );
      });
   });
});
```

以上代码执行结果如下：

```
$ node file.js
准备删除目录 /tmp/test
input.out
output.out
test
test.txt
读取 /tmp 目录
……
```

文件模块方法参考手册

以下为 Node.js 文件模块相同的方法列表：

方法	描述
fs.rename(oldPath, newPath, callback)	异步 rename().回调函数没有参数，但可能抛出异常。
fs.ftruncate(fd, len, callback)	异步 ftruncate().回调函数没有参数，但可能抛出异常。
fs.ftruncateSync(fd, len)	同步 ftruncate()

fs.truncate(path, len, callback)	异步 truncate().回调函数没有参数，但可能抛出异常。
fs.truncateSync(path, len)	同步 truncate()
fs.chown(path, uid, gid, callback)	异步 chown().回调函数没有参数，但可能抛出异常。
fs.chownSync(path, uid, gid)	同步 chown()
fs.fchown(fd, uid, gid, callback)	异步 fchown().回调函数没有参数，但可能抛出异常。
fs.fchownSync(fd, uid, gid)	同步 fchown()
fs.lchown(path, uid, gid, callback)	异步 lchown().回调函数没有参数，但可能抛出异常。
fs.lchownSync(path, uid, gid)	同步 lchown()
fs.chmod(path, mode, callback)	异步 chmod().回调函数没有参数，但可能抛出异常。
fs.chmodSync(path, mode)	同步 chmod().
fs.fchmod(fd, mode, callback)	异步 fchmod().回调函数没有参数，但可能抛出异常。
fs.fchmodSync(fd, mode)	同步 fchmod().
fs.lchmod(path, mode, callback)	异步 lchmod().回调函数没有参数，但可能抛出异常。Only available on Mac OS X.
fs.lchmodSync(path, mode)	同步 lchmod().
fs.stat(path, callback)	异步 stat(). 回调函数有两个参数 err, stats，stats 是 fs.Stats 对象。
fs.lstat(path, callback)	异步 lstat(). 回调函数有两个参数 err, stats，stats 是 fs.Stats 对象。
fs.fstat(fd, callback)	异步 fstat(). 回调函数有两个参数 err, stats，stats 是 fs.Stats 对象。
fs.statSync(path)	同步 stat(). 返回 fs.Stats 的实例。
fs.lstatSync(path)	同步 lstat(). 返回 fs.Stats 的实例。
fs.fstatSync(fd)	同步 fstat(). 返回 fs.Stats 的实例。
fs.link(srcpath, dstpath, callback)	异步 link().回调函数没有参数，但可能抛出异常。
fs.linkSync(srcpath, dstpath)	同步 link().
fs.symlink(srcpath,	异步 symlink().回调函数没有参数，但可能抛出异常。type 参数可以设置为 'dir', 'file', 或 'junction' (默认为

	'file') 。
fs.symlinkSync(srcpath, dstpath[, type])	同步 symlink().
fs.readlink(path, callback)	异步 readlink(). 回调函数有两个参数 err, linkString。
fs.realpath(path[, cache], callback)	异步 realpath(). 回调函数有两个参数 err, resolvedPath。
fs.realpathSync(path[, cache])	同步 realpath()。返回绝对路径。
fs.unlink(path, callback)	异步 unlink().回调函数没有参数，但可能抛出异常。
fs.unlinkSync(path)	同步 unlink().
fs.rmdir(path, callback)	异步 rmdir().回调函数没有参数，但可能抛出异常。
fs.rmdirSync(path)	同步 rmdir().
fs.mkdir(path[, mode], callback)	S异步 mkdir(2).回调函数没有参数，但可能抛出异常。 mode defaults to 0777.
fs.mkdirSync(path[, mode])	同步 mkdir().
fs.readdir(path, callback)	异步 readdir(3). 读取目录的内容。
fs.readdirSync(path)	同步 readdir().返回文件数组列表。
fs.close(fd, callback)	异步 close().回调函数没有参数，但可能抛出异常。
fs.closeSync(fd)	同步 close().
fs.open(path, flags[, mode], callback)	异步打开文件。
fs.openSync(path, flags[, mode])	同步 version of fs.open().
fs.utimes(path, atime, mtime, callback)	?
fs.utimesSync(path, atime, mtime)	修改文件时间戳，文件通过指定的文件路径。
fs.futimes(fd, atime, mtime, callback)	?
fs.futimesSync(fd, atime, mtime)	修改文件时间戳，通过文件描述符指定。
fs.fsync(fd, callback)	异步 fsync.回调函数没有参数，但可能抛出异常。
fs.fsyncSync(fd)	同步 fsync.
fs.write(fd, buffer, offset, length[, position], callback)	将缓冲区内容写入到通过文件描述符指定的文件。
fs.write(fd, data[, position[,	通过文件描述符 fd 写入文件内容。

encoding]], callback)	通过文件描述符 fd 写入文件内容。
fs.writeSync(fd, buffer, offset, length[, position])	同步版的 fs.write()。
fs.writeSync(fd, data[, position[, encoding]])	同步版的 fs.write().
fs.read(fd, buffer, offset, length, position, callback)	通过文件描述符 fd 读取文件内容。
fs.readSync(fd, buffer, offset, length, position)	同步版的 fs.read.
fs.readFile(filename[, options], callback)	异步读取文件内容。
fs.readFileSync(filename[, options])	
fs.writeFile(filename, data[, options], callback)	异步写入文件内容。
fs.writeFileSync(filename, data[, options])	同步版的 fs.writeFile。
fs.appendFile(filename, data[, options], callback)	异步追加文件内容。
fs.appendFileSync(filename, data[, options])	The 同步 version of fs.appendFile.
fs.watchFile(filename[, options], listener)	查看文件的修改。
fs.unwatchFile(filename[, listener])	停止查看 filename 的修改。
fs.watch(filename[, options][, listener])	查看 filename 的修改，filename 可以是文件或目录。返回 fs.FSWatcher 对象。
fs.exists(path, callback)	检测给定的路径是否存在。
fs.existsSync(path)	同步版的 fs.exists.
fs.access(path[, mode], callback)	测试指定路径用户权限。
fs.accessSync(path[, mode])	同步版的 fs.access。
fs.createReadStream(path[, options])	返回ReadStream 对象。
fs.createWriteStream(path[, options])	返回 WriteStream 对象。
fs.symlink(srcpath,	异步 symlink().回调函数没有参数，但可能抛出异常。

更多内容，请查看官网文件模块描述：File System。

Node.js GET/POST请求

在很多场景中，我们的服务器都需要跟用户的浏览器打交道，如表单提交。

表单提交到服务器一般都使用GET/POST请求。

本章节我们将为大家介绍 Node.js GET/POST请求。

获取GET请求内容

由于GET请求直接被嵌入在路径中，URL是完整的请求路径，包括了?后面的部分，因此你可以手动解析后面的内容作为GET请求的参数。

node.js中url模块中的parse函数提供了这个功能。

```
var http = require('http');
var url = require('url');
var util = require('util');

http.createServer(function(req, res){
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end(util.inspect(url.parse(req.url, true)));
}).listen(3000);
```

在浏览器中访问 `http://localhost:3000/user?name=w3c&email=w3c@w3cschool.cc` 然后查看返回结果:

```
{ protocol: null,
  slashes: null,
  auth: null,
  host: null,
  port: null,
  hostname: null,
  hash: null,
  search: '?name=w3c&email=w3c@w3cschool.cc',
  query: { name: 'w3c', email: 'w3c@w3cschool.cc' },
  pathname: '/user',
  path: '/user?name=w3c&email=w3c@w3cschool.cc',
  href: '/user?name=w3c&email=w3c@w3cschool.cc' }
```

获取POST请求内容

POST请求的内容全部的都在请求体中，http.ServerRequest并没有一个属性内容为请求体，原因是等待请求体传输可能是一件耗时的工作。

比如上传文件，而很多时候我们可能并不需要理会请求体的内容，恶意的POST请求会大大消耗服务器的资源，所有node.js默认是不会解析请求体的， 当你需要的时候，需要手动来做。

```
var http = require('http');
var querystring = require('querystring');
var util = require('util');

http.createServer(function(req, res){
    var post = '';     //定义了一个post变量，用于暂存请求体的信息

    req.on('data', function(chunk){    //通过req的data事件监听函数，每当接受到请求体的数据，就累加
        post += chunk;
    });

    req.on('end', function(){    //在end事件触发后，通过querystring.parse将post解析为真正的POST
        post = querystring.parse(post);
        res.end(util.inspect(post));
    });
}).listen(3000);
```

Node.js 工具模块

在 Node.js 模块库中有很多好用的模块。接下来我们为大家介绍几种常用模块的使用：

模块名	描述
OS 模块	提供基本的系统操作函数。
Path 模块	提供了处理和转换文件路的工具。
Net 模块	用于底层的网络通信。提供了服务端和客户端的的操作。
DNS 模块	用于解析域名。
Domain 模块	简化异步代码的异常处理，可以捕捉处理try catch无法捕捉的。

Node.js OS 模块

Node.js os 模块提供了一些基本的系统操作函数。我们可以通过以下方式引入该模块：

```
var os =  require("os")
```

方法

方法	描述
os.tmpdir()	返回操作系统的默认临时文件夹。
os.endianness()	返回 CPU 的字节序，可能的是 "BE" 或 "LE"。
os.hostname()	返回操作系统的主机名。
os.type()	返回操作系统名
os.platform()	返回操作系统名
os.arch()	返回操作系统 CPU 架构，可能的值有 "x64"、"arm" 和 "ia32"。
os.release()	返回操作系统的发行版本。
os.uptime()	返回操作系统运行的时间，以秒为单位。
os.loadavg()	返回一个包含 1、5、15 分钟平均负载的数组。
os.totalmem()	返回系统内存总量，单位为字节。
os.freemem()	返回操作系统空闲内存量，单位是字节。
os.cpus()	返回一个对象数组，包含所安装的每个 CPU/内核的信息：型号、速度（单位 MHz）、时间（一个包含 user、nice、sys、idle 和 irq 所使用 CPU/内核毫秒数的对象）。
os.networkInterfaces()	获得网络接口列表。

属性

属性	描述
os.EOL	定义了操作系统的行尾符的常量。

实例

创建 main.js 文件，代码如下所示：

```
var os =  require("os");  // CPU 的字节序 console.log('endianness : '  + os.endianness());
```

代码执行结果如下：

```
$ node main.js
endianness : LE
type :  Linux platform : linux
total memory :  25103400960 bytes. free memory :  20676710400 bytes.
```

Node.js Path 模块

Node.js path 模块提供了一些用于处理文件路径的小工具，我们可以通过以下方式引入该模块：

```
var path = require("path")
```

方法

方法	描述
path.normalize(p)	规范化路径，注意'..' 和 '.'。
path.join([path1][, path2][, ...])	用于连接路径。该方法的主要用途在于，会正确使用当前系统的路径分隔符，Unix系统是"/"，Windows系统是"\"。
path.resolve([from ...], to)	将 **to** 参数解析为绝对路径。
path.isAbsolute(path)	判断参数 **path** 是否是绝对路径。
path.relative(from, to)	用于将相对路径转为绝对路径。
path.dirname(p)	返回路径中代表文件夹的部分，同 Unix 的dirname 命令类似。
path.basename(p[, ext])	返回路径中的最后一部分。同 Unix 命令 bashname 类似。
path.extname(p)	返回路径中文件的后缀名，即路径中最后一个'.'之后的部分。如果一个路径中并不包含'.'或该路径只包含一个'.' 且这个'.'为路径的第一个字符，则此命令返回空字符串。
path.parse(pathString)	返回路径字符串的对象。
path.format(pathObject)	从对象中返回路径字符串，和 path.parse 相反。

属性

属性	描述
path.sep	平台的文件路径分隔符，'\' 或 '/'。
path.delimiter	平台的分隔符, ; or ':'.
path.posix	提供上述 path 的方法，不过总是以 posix 兼容的方式交互。
path.win32	提供上述 path 的方法，不过总是以 win32 兼容的方式交互。

实例

创建 main.js 文件，代码如下所示：

```
var path = require("path");

// 格式化路径
console.log('normalization : ' + path.normalize('/test/test1//2slashes/1slash/tab/..'));

// 连接路径
console.log('joint path : ' + path.join('/test', 'test1', '2slashes/1slash', 'tab', '..')

// 转换为绝对路径
console.log('resolve : ' + path.resolve('main.js'));

// 路径中文件的后缀名
console.log('ext name : ' + path.extname('main.js'));
```

代码执行结果如下：

```
$ node main.js
normalization : /test/test1/2slashes/1slash
joint path : /test/test1/2slashes/1slash
resolve : /web/com/1427176256_27423/main.js
ext name : .js
```

Node.js Net 模块

Node.js Net 模块提供了一些用于底层的网络通信的小工具，包含了创建服务器/客户端的方法，我们可以通过以下方式引入该模块：

```
var net = require("net")
```

方法

方法	描述
net.createServer([options][, connectionListener])	创建一个 TCP 服务器。参数 connectionListener 自动给 'connection' 事件创建监听器。
net.connect(options[, connectionListener])	返回一个新的 'net.Socket'，并连接到指定的地址和端口。 当 socket 建立的时候，将会触发 'connect' 事件。
net.createConnection(options[, connectionListener])	创建一个到端口 port 和 主机 host的 TCP 连接。 host 默认为 'localhost'。
net.connect(port[, host][, connectListener])	创建一个端口为 port 和主机为 host的 TCP 连接 。host 默认为 'localhost'。参数 connectListener 将会作为监听器添加到 'connect' 事件。返回 'net.Socket'。
net.createConnection(port[, host][, connectListener])	创建一个端口为 port 和主机为 host的 TCP 连接 。host 默认为 'localhost'。参数 connectListener 将会作为监听器添加到 'connect' 事件。返回 'net.Socket'。
net.connect(path[, connectListener])	创建连接到 path 的 unix socket 。参数 connectListener 将会作为监听器添加到 'connect' 事件上。返回 'net.Socket'。
net.createConnection(path[, connectListener])	创建连接到 path 的 unix socket 。参数 connectListener 将会作为监听器添加到 'connect' 事件。返回 'net.Socket'。
net.isIP(input)	检测输入的是否为 IP 地址。 IPV4 返回 4， IPV6 返回 6，其他情况返回 0。
net.isIPv4(input)	如果输入的地址为 IPV4， 返回 true，否则返回 false。
net.isIPv6(input)	如果输入的地址为 IPV6， 返回 true，否则返回 false。

net.Server

net.Server通常用于创建一个 TCP 或本地服务器。

方法	描述
server.listen(port[, host][, backlog][, callback])	监听指定端口 port 和 主机 host ac连接。 默认情况下 host 接受任何 IPv4 地址(INADDR_ANY)的直接连接。端口 port 为 0 时，则会分配一个随机端口。
server.listen(path[, callback])	通过指定 path 的连接， 启动一个本地 socket 服务器。
server.listen(handle[, callback])	通过指定句柄连接。
server.listen(options[, callback])	options 的属性：端口 port, 主机 host, 和 backlog, 以及可选参数 callback 函数, 他们在一起调用server.listen(port, [host], [backlog], [callback])。还有， 参数 path 可以用来指定 UNIX socket。
server.close([callback])	服务器停止接收新的连接，保持现有连接。这是异步函数， 当所有连接结束的时候服务器会关闭，并会触发 'close' 事件。
server.address()	操作系统返回绑定的地址， 协议族名和服务器端口。
server.unref()	如果这是事件系统中唯一一个活动的服务器， 调用 unref 将允许程序退出。
server.ref()	与 unref 相反， 如果这是唯一的服务器， 在之前被 unref 了的服务器上调用 ref 将不会让程序退出（默认行为）。如果服务器已经被 ref， 则再次调用 ref 并不会产生影响。
server.getConnections(callback)	异步获取服务器当前活跃连接的数量。当 socket 发送给子进程后才有效；回调函数有 2 个参数 err 和 count。

事件

事件	描述
listening	当服务器调用 server.listen 绑定后会触发。
connection	当新连接创建后会被触发。socket 是 net.Socket实例。
close	服务器关闭时会触发。注意，如果存在连接， 这个事件不会被触发直到所有的连接关闭。
error	发生错误时触发。'close' 事件将被下列事件直接调用。

net.Socket

net.Socket 对象是 TCP 或 UNIX Socket 的抽象。net.Socket 实例实现了一个双工流接口。他们可以在用户创建客户端(使用 connect())时使用, 或者由 Node 创建它们，并通过 connection 服务器事件传递给用户。

事件

net.Socket 事件有：

事件	描述
lookup	在解析域名后，但在连接前，触发这个事件。对 UNIX sokcet 不适用。
connect	成功建立 socket 连接时触发。
data	当接收到数据时触发。
end	当 socket 另一端发送 FIN 包时，触发该事件。
timeout	当 socket 空闲超时时触发，仅是表明 socket 已经空闲。用户必须手动关闭连接。
drain	当写缓存为空得时候触发。可用来控制上传。
error	错误发生时触发。
close	当 socket 完全关闭时触发。参数 had_error 是布尔值，它表示是否因为传输错误导致 socket 关闭。

属性

net.Socket 提供了很多有用的属性，便于控制 socket 交互：

属性	描述
socket.bufferSize	该属性显示了要写入缓冲区的字节数。
socket.remoteAddress	远程的 IP 地址字符串，例如：'74.125.127.100' or '2001:4860:a005::68'。
socket.remoteFamily	远程IP协议族字符串，比如 'IPv4' or 'IPv6'。
socket.remotePort	远程端口，数字表示，例如：80 or 21。
socket.localAddress	网络连接绑定的本地接口 远程客户端正在连接的本地 IP 地址，字符串表示。例如，如果你在监听'0.0.0.0'而客户端连接在'192.168.1.1'，这个值就会是 '192.168.1.1'。
socket.localPort	本地端口地址，数字表示。例如：80 or 21。
socket.bytesRead	接收到得字节数。
socket.bytesWritten	发送的字节数。

方法

方法	描述
new net.Socket([options])	构造一个新的 socket 对象。
socket.connect(port[, host][, connectListener])	指定端口 port 和 主机 host，创建 socket 连接 。参数 host 默认为 localhost。通常情况不需要使用 net.createConnection 打开 socket。只有你实现了自己的 socket 时才会用到。
socket.connect(path[, connectListener])	打开指定路径的 unix socket。通常情况不需要使用 net.createConnection 打开 socket。只有你实现了自己的 socket 时才会用到。
socket.setEncoding([encoding])	设置编码
socket.write(data[, encoding][, callback])	在 socket 上发送数据。第二个参数指定了字符串的编码，默认是 UTF8 编码。
socket.end([data][, encoding])	半关闭 socket。例如，它发送一个 FIN 包。可能服务器仍在发送数据。
socket.destroy()	确保没有 I/O 活动在这个套接字上。只有在错误发生情况下才需要。（处理错误等等）。
socket.pause()	暂停读取数据。就是说，不会再触发 data 事件。对于控制上传非常有用。
socket.resume()	调用 pause() 后想恢复读取数据。
socket.setTimeout(timeout[, callback])	socket 闲置时间超过 timeout 毫秒后 ，将 socket 设置为超时。
socket.setNoDelay([noDelay])	禁用纳格（Nagle）算法。默认情况下 TCP 连接使用纳格算法，在发送前他们会缓冲数据。将 noDelay 设置为 true 将会在调用 socket.write() 时立即发送数据。noDelay 默认值为 true。
socket.setKeepAlive([enable][, initialDelay])	禁用/启用长连接功能，并在发送第一个在闲置 socket 上的长连接 probe 之前，可选地设定初始延时。默认为 false。 设定 initialDelay （毫秒），来设定收到的最后一个数据包和第一个长连接 probe之间的延时。将 initialDelay 设为0，将会保留默认（或者之前）的值。默认值为0.
socket.address()	操作系统返回绑定的地址，协议族名和服务器端口。返回的对象有 3 个属性，比如{ port: 12346, family: 'IPv4', address: '127.0.0.1' }。
socket.unref()	如果这是事件系统中唯一一个活动的服务器，调用 unref 将允许程序退出。如果服务器已被 unref，则再次调用 unref 并不会产生影响。
socket.ref()	与 unref 相反，如果这是唯一的服务器，在之前被 unref 了的服务器上调用 ref 将不会让程序退出（默认行为）。如果服务器已经被 ref，则再次调用 ref 并不会产生影响。

实例

创建 server.js 文件，代码如下所示：

```
var net = require('net');
var server = net.createServer(function(connection) {
   console.log('client connected');
   connection.on('end', function() {
      console.log('客户端关闭连接');
   });
   connection.write('Hello World!\r\n');
   connection.pipe(connection);
});
server.listen(8080, function() {
  console.log('server is listening');
});
```

执行以上服务端代码：

```
$ node server.js
server is listening   # 服务已创建并监听 8080 端口
```

新开一个窗口，创建 client.js 文件，代码如下所示：

```
var net = require('net');
var client = net.connect({port: 8080}, function() {
   console.log('连接到服务器！');
});
client.on('data', function(data) {
   console.log(data.toString());
   client.end();
});
client.on('end', function() {
   console.log('断开与服务器的连接');
});
```

执行以上客户端的代码：

```
连接到服务器！
Hello World!

断开与服务器的连接
```

Gif 实例演示

node — vim — 80×24

```
var net = require('net');
var server = net.createServer(function(connection) {
   console.log('client connected');
   connection.on('end', function() {
      console.log('客户端关闭连接');
   });
   connection.write('Hello World!\r\n');
   connection.pipe(connection);
});
server.listen(8080, function() {
  console.log('server is listening');
});
~
~
~
~
~
~
~
~
~
~
~
"server.js" 12L, 357C
```

Node.js DNS 模块

Node.js **DNS** 模块用于解析域名。引入 DNS 模块语法格式如下：

```
var dns = require("dns")
```

方法

方法	描述
dns.lookup(hostname[, options], callback)	将域名（比如 'runoob.com'）解析为第一条找到的记录 A （IPV4）或 AAAA(IPV6)。参数 options可以是一个对象或整数。如果没有提供 options，IP v4 和 v6 地址都可以。如果 options 是整数，则必须是 4 或 6。
dns.lookupService(address, port, callback)	使用 getnameinfo 解析传入的地址和端口为域名和服务。
dns.resolve(hostname[, rrtype], callback)	将一个域名（如 'runoob.com'）解析为一个 rrtype 指定记录类型的数组。
dns.resolve4(hostname, callback)	和 dns.resolve() 类似, 仅能查询 IPv4 (A 记录）。 addresses IPv4 地址数组 (比如，['74.125.79.104', '74.125.79.105', '74.125.79.106']）。
dns.resolve6(hostname, callback)	和 dns.resolve4() 类似， 仅能查询 IPv6(AAAA 查询）
dns.resolveMx(hostname, callback)	和 dns.resolve() 类似, 仅能查询邮件交换(MX 记录)。
dns.resolveTxt(hostname, callback)	和 dns.resolve() 类似, 仅能进行文本查询 (TXT 记录）。 addresses 是 2-d 文本记录数组。(比如，[['v=spf1 ip4:0.0.0.0 ', '~all']]）。 每个子数组包含一条记录的 TXT 块。根据使用情况可以连接在一起，也可单独使用。
dns.resolveSrv(hostname, callback)	和 dns.resolve() 类似, 仅能进行服务记录查询 (SRV 记录）。 addresses 是 hostname可用的 SRV 记录数组。 SRV 记录属性有优先级（priority），权重（weight）, 端口（port）, 和名字（name） (比如，[{'priority': 10, 'weight': 5, 'port': 21223, 'name': 'service.example.com'}, ...]）。
dns.resolveSoa(hostname, callback)	和 dns.resolve() 类似, 仅能查询权威记录(SOA 记录）。
dns.resolveNs(hostname, callback)	和 dns.resolve() 类似, 仅能进行域名服务器记录查询(NS 记录）。 addresses 是域名服务器记录数组（hostname 可以使用） (比如, ['ns1.example.com', 'ns2.example.com']）。
dns.resolveCname(hostname, callback)	和 dns.resolve() 类似, 仅能进行别名记录查询 (CNAME记录)。addresses 是对 hostname 可用的别名记录数组 (比如，, ['bar.example.com']）。
dns.reverse(ip, callback)	反向解析 IP 地址，指向该 IP 地址的域名数组。
dns.getServers()	返回一个用于当前解析的 IP 地址数组的字符串。
dns.setServers(servers)	指定一组 IP 地址作为解析服务器。

rrtypes

以下列出了 dns.resolve() 方法中有效的 rrtypes值:

- `'A'` IPV4 地址, 默认
- `'AAAA'` IPV6 地址
- `'MX'` 邮件交换记录
- `'TXT'` text 记录
- `'SRV'` SRV 记录
- `'PTR'` 用来反向 IP 查找
- `'NS'` 域名服务器记录
- `'CNAME'` 别名记录
- `'SOA'` 授权记录的初始值

错误码

每次 DNS 查询都可能返回以下错误码：

- `dns.NODATA`：无数据响应。
- `dns.FORMERR`：查询格式错误。
- `dns.SERVFAIL`：常规失败。
- `dns.NOTFOUND`：没有找到域名。
- `dns.NOTIMP`：未实现请求的操作。
- `dns.REFUSED`：拒绝查询。
- `dns.BADQUERY`：查询格式错误。
- `dns.BADNAME`：域名格式错误。
- `dns.BADFAMILY`：地址协议不支持。
- `dns.BADRESP`：回复格式错误。
- `dns.CONNREFUSED`：无法连接到 DNS 服务器。
- `dns.TIMEOUT`：连接 DNS 服务器超时。
- `dns.EOF`：文件末端。
- `dns.FILE`：读文件错误。
- `dns.NOMEM`：内存溢出。
- `dns.DESTRUCTION`：通道被摧毁。
- `dns.BADSTR`：字符串格式错误。
- `dns.BADFLAGS`：非法标识符。
- `dns.NONAME`：所给主机不是数字。
- `dns.BADHINTS`：非法HINTS标识符。
- `dns.NOTINITIALIZED`：c c-ares 库尚未初始化。
- `dns.LOADIPHLPAPI`：加载 iphlpapi.dll 出错。
- `dns.ADDRGETNETWORKPARAMS`：无法找到 GetNetworkParams 函数。
- `dns.CANCELLED`：取消 DNS 查询。

实例

创建 main.js 文件，代码如下所示：

```
var dns = require('dns');

dns.lookup('www.github.com', function onLookup(err, address, family) {
   console.log('ip 地址:', address);
   dns.reverse(address, function (err, hostnames) {
   if (err) {
      console.log(err.stack);
   }

   console.log('反向解析 ' + address + ': ' + JSON.stringify(hostnames));
});
});
```

执行以上代码，结果如下所示:

```
address: 192.30.252.130
reverse for 192.30.252.130: ["github.com"]
```

Node.js Domain 模块

Node.js **Domain(域)** 简化异步代码的异常处理，可以捕捉处理try catch无法捕捉的异常。引入 Domain 模块 语法格式如下：

```
var domain = require("domain")
```

domain模块，把处理多个不同的IO的操作作为一个组。注册事件和回调到domain，当发生一个错误事件或抛出一个错误时，domain对象会被通知，不会丢失上下文环境，也不导致程序错误立即推出，与process.on('uncaughtException')不同。

Domain 模块可分为隐式绑定和显式绑定：

- 隐式绑定: 把在domain上下文中定义的变量，自动绑定到domain对象
- 显式绑定: 把不是在domain上下文中定义的变量，以代码的方式绑定到domain对象

方法

方法	描述
domain.run(function)	在域的上下文运行提供的函数，隐式的绑定了所有的事件分发器，计时器和底层请求。
domain.add(emitter)	显式的增加事件
domain.remove(emitter)	删除事件。
domain.bind(callback)	返回的函数是一个对于所提供的回调函数的包装函数。当调用这个返回的函数被时，所有被抛出的错误都会被导向到这个域的 error 事件。
domain.intercept(callback)	和 domain.bind(callback) 类似。除了捕捉被抛出的错误外，它还会拦截 Error 对象作为参数传递到这个函数。
domain.enter()	进入一个异步调用的上下文，绑定到domain。
domain.exit()	退出当前的domain，切换到不同的链的异步调用的上下文中。对应domain.enter()。
domain.dispose()	释放一个domain对象，让node进程回收这部分资源。
domain.create()	返回一个domain对象。

属性

属性	描述
domain.members	已加入domain对象的域定时器和事件发射器的数组。

实例

创建 main.js 文件，代码如下所示：

```
var EventEmitter = require("events").EventEmitter;
var domain = require("domain");

var emitter1 = new EventEmitter();

// 创建域
var domain1 = domain.create();

domain1.on('error', function(err){
   console.log("domain1 处理这个错误 ("+err.message+")");
});

// 显式绑定
domain1.add(emitter1);

emitter1.on('error',function(err){
   console.log("监听器处理此错误 ("+err.message+")");
});

emitter1.emit('error',new Error('通过监听器来处理'));

emitter1.removeAllListeners('error');

emitter1.emit('error',new Error('通过 domain1 处理'));

var domain2 = domain.create();

domain2.on('error', function(err){
   console.log("domain2 处理这个错误 ("+err.message+")");
});

// 隐式绑定
domain2.run(function(){
   var emitter2 = new EventEmitter();
   emitter2.emit('error',new Error('通过 domain2 处理'));
});

domain1.remove(emitter1);
emitter1.emit('error', new Error('转换为异常，系统将崩溃!'));
```

执行以上代码，结果如下所示:

```
监听器处理此错误 (通过监听器来处理)
domain1 处理这个错误 (通过 domain1 处理)
domain2 处理这个错误 (通过 domain2 处理)

events.js:72
        throw er; // Unhandled 'error' event
              ^
Error: 转换为异常，系统将崩溃!
    at Object.<anonymous> (/www/node/main.js:40:24)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:929:3
```

Node.js Web 模块

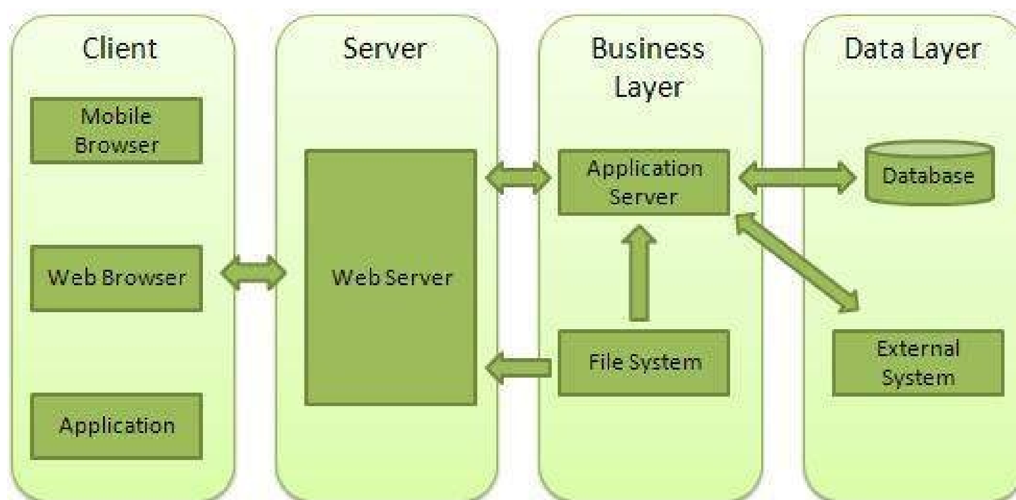
什么是 Web 服务器？

Web服务器一般指网站服务器，是指驻留于因特网上某种类型计算机的程序，Web服务器的基本功能就是提供Web信息浏览服务。它只需支持HTTP协议、HTML文档格式及URL，与客户端的网络浏览器配合。

大多数 web 服务器都支持服务端的脚本语言（php、python、ruby）等，并通过脚本语言从数据库获取数据，将结果返回给客户端浏览器。

目前最主流的三个Web服务器是Apache、Nginx、IIS。

Web 应用架构



- **Client** - 客户端，一般指浏览器，浏览器可以通过 HTTP 协议向服务器请求数据。
- **Server** - 服务端，一般指 Web 服务器，可以接收客户端请求，并向客户端发送响应数据。
- **Business** - 业务层， 通过 Web 服务器处理应用程序，如与数据库交互，逻辑运算，调用外部程序等。
- **Data** - 数据层，一般由数据库组成。

使用 Node 创建 Web 服务器

Node.js 提供了 http 模块，http 模块主要用于搭建 HTTP 服务端和客户端，使用 HTTP 服务器或客户端功能必须调用 http 模块，代码如下：

```
var http = require('http');
```

以下是演示一个最基本的 HTTP 服务器架构(使用8081端口)，创建 server.js 文件，代码如下所示：

```
var http = require('http');
var fs = require('fs');
var url = require('url');

// 创建服务器
http.createServer( function (request, response) {
   // 解析请求，包括文件名
   var pathname = url.parse(request.url).pathname;

   // 输出请求的文件名
   console.log("Request for " + pathname + " received.");

   // 从文件系统中读取请求的文件内容
   fs.readFile(pathname.substr(1), function (err, data) {
      if (err) {
         console.log(err);
         // HTTP 状态码: 404 : NOT FOUND
         // Content Type: text/plain
         response.writeHead(404, {'Content-Type': 'text/html'});
      }else{
         // HTTP 状态码: 200 : OK
         // Content Type: text/plain
         response.writeHead(200, {'Content-Type': 'text/html'});

         // 响应文件内容
         response.write(data.toString());
      }
      //  发送响应数据
      response.end();
   });
}).listen(8081);

// 控制台会输出以下信息
console.log('Server running at http://127.0.0.1:8081/');
```

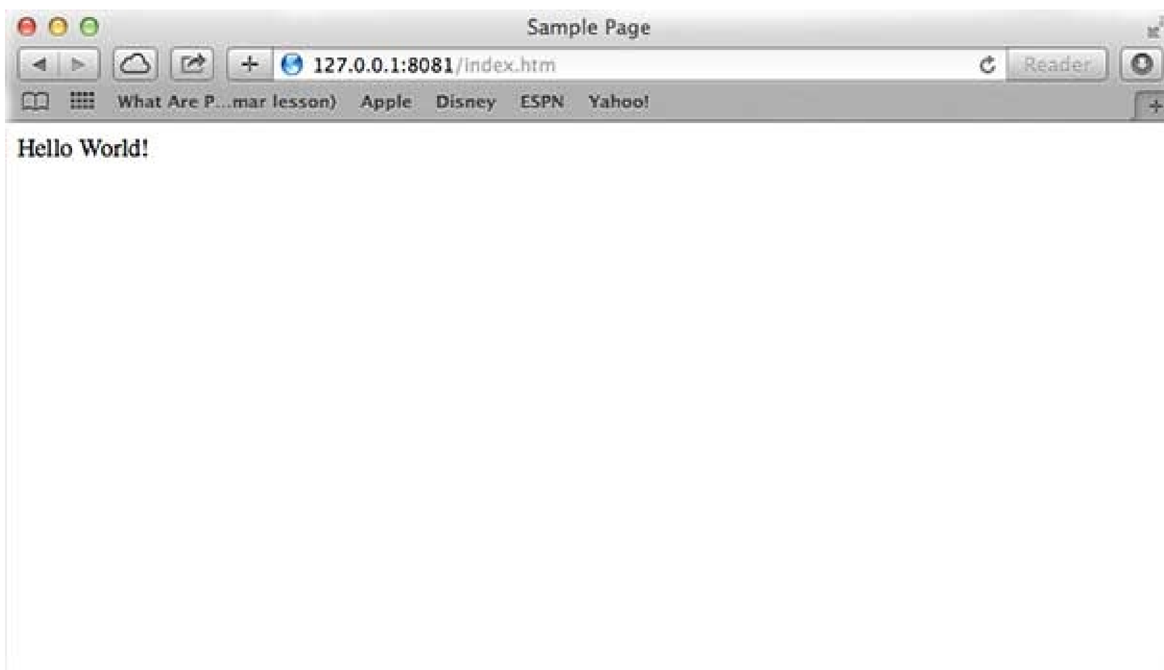
接下来我们在该目录下创建一个 index.htm 文件，代码如下：

```
<html>
<head>
<title>Sample Page</title>
</head>
<body>
Hello World!
</body>
</html>
```

执行 server.js 文件：

```
$ node server.js
Server running at http://127.0.0.1:8081/
```

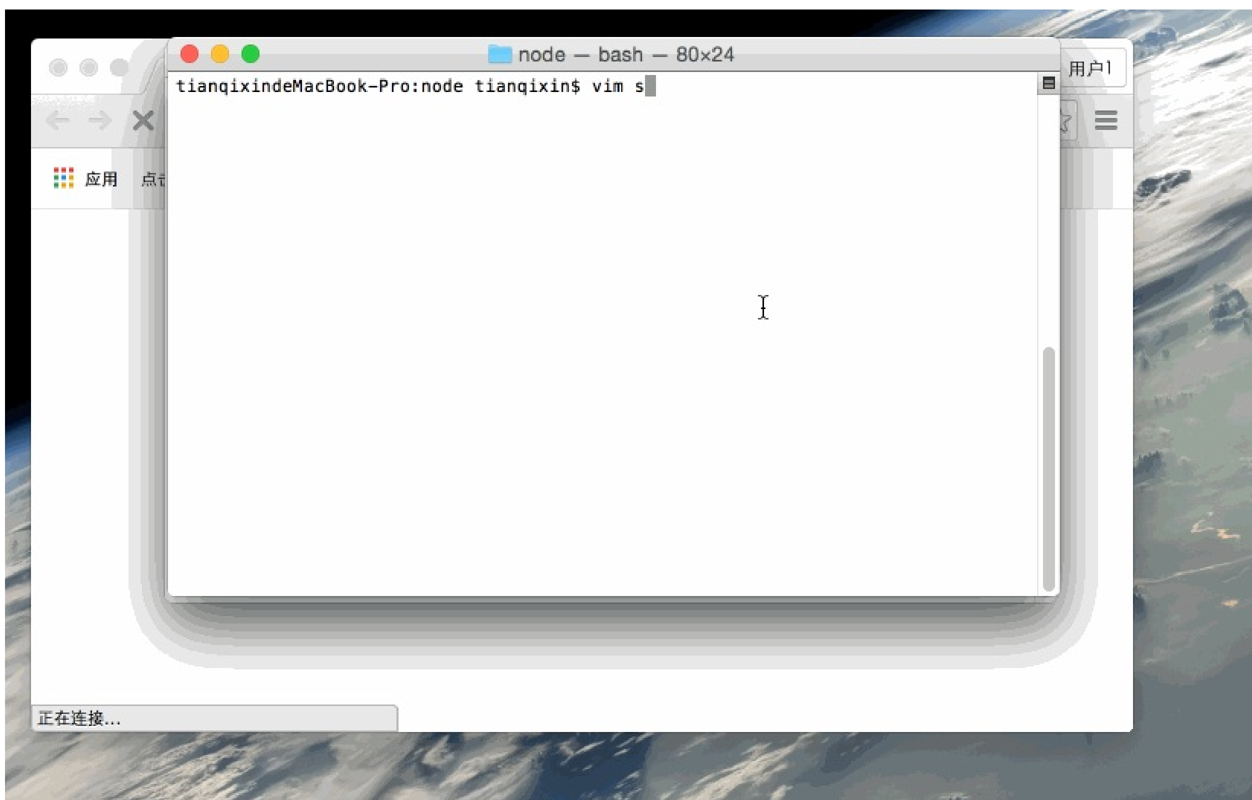
接着我们在浏览器中打开地址：http://127.0.0.1:8081/index.htm，显示如下图所示:



执行 server.js 的控制台输出信息如下：

```
Server running at http://127.0.0.1:8081/
Request for /index.htm received.     #  客户端请求信息
```

Gif 实例演示



使用 Node 创建 Web 客户端

Node 创建 Web 客户端需要引入 http 模块，创建 client.js 文件，代码如下所示：

```
<pre>
var http = require('http');

// 用于请求的选项
var options = {
   host: 'localhost',
   port: '8081',
   path: '/index.htm'
};

// 处理响应的回调函数
var callback = function(response){
   // 不断更新数据
   var body = '';
   response.on('data', function(data) {
      body += data;
   });

   response.on('end', function() {
      // 数据接收完成
      console.log(body);
   });
}
// 向服务端发送请求
var req = http.request(options, callback);
req.end();
```

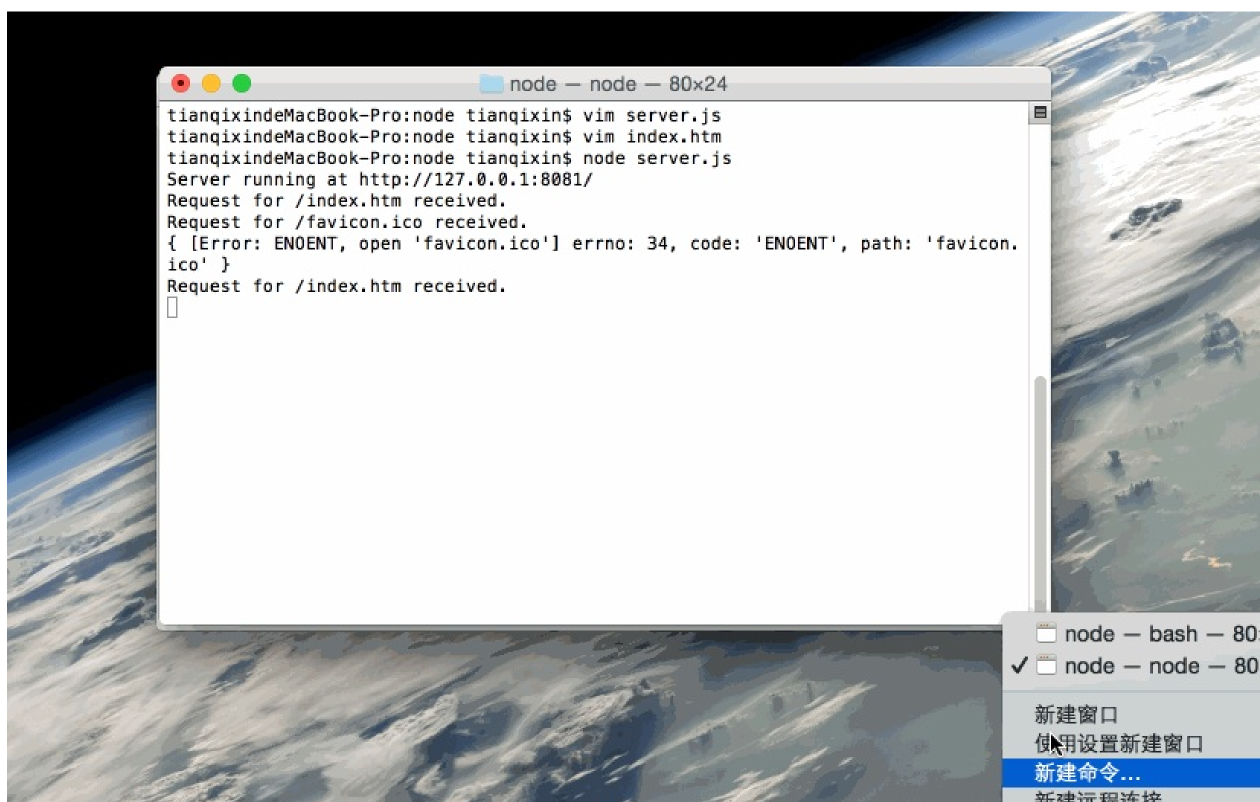
新开一个终端，执行 client.js 文件，输出结果如下：

```
$ node client.js
<html>
<head>
<title>Sample Page</title>
</head>
<body>
Hello World!
</body>
</html>
```

执行 server.js 的控制台输出信息如下：

```
Server running at http://127.0.0.1:8081/
Request for /index.htm received.   # 客户端请求信息
```

Gif 实例演示


node — node — 80×24
tianqixindeMacBook-Pro:node tianqixin$ vim server.js
tianqixindeMacBook-Pro:node tianqixin$ vim index.htm
tianqixindeMacBook-Pro:node tianqixin$ node server.js
Server running at http://127.0.0.1:8081/
Request for /index.htm received.
Request for /favicon.ico received.
{ [Error: ENOENT, open 'favicon.ico'] errno: 34, code: 'ENOENT', path: 'favicon.ico' }
Request for /index.htm received.
新建窗口
新建命令...

Node.js Express 框架

Express 简介

Express 是一个简洁而灵活的 node.js Web应用框架, 提供了一系列强大特性帮助你创建各种 Web 应用，和丰富的 HTTP 工具。

使用 Express 可以快速地搭建一个完整功能的网站。

Express 框架核心特性：

- 可以设置中间件来响应 HTTP 请求。
- 定义了路由表用于执行不同的 HTTP 请求动作。
- 可以通过向模板传递参数来动态渲染 HTML 页面。

安装 Express

安装 Express 并将其保存到依赖列表中：

```
$ npm install express --save
```

以上命令会将 Express 框架安装在当期目录的 **node_modules** 目录中， **node_modules** 目录下会自动创建 express 目录。以下几个重要的模块是需要与 express 框架一起安装的：

- **body-parser** - node.js 中间件，用于处理 JSON, Raw, Text 和 URL 编码的数据。
- **cookie-parser** - 这就是一个解析Cookie的工具。通过req.cookies可以取到传过来的cookie，并把它们转成对象。
- **multer** - node.js 中间件，用于处理 enctype="multipart/form-data"（设置表单的MIME编码）的表单数据。

```
$ npm install body-parser --save
$ npm install cookie-parser --save
$ npm install multer --save
```

第一个 Express 框架实例

接下来我们使用 Express 框架来输出 "Hello World"。

以下实例中我们引入了 express 模块，并在客户端发起请求后，响应 "Hello World" 字符串。

创建 express_demo.js 文件，代码如下所示：

```
//express_demo.js 文件
var express = require('express');
var app = express();

app.get('/', function (req, res) {
   res.send('Hello World');
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

执行以上代码：

```
$ node express_demo.js
应用实例，访问地址为 http://0.0.0.0:8081
```

在浏览器中访问 http://127.0.0.1:8081，结果如下图所示：



请求和响应

Express 应用使用回调函数的参数： **request** 和 **response** 对象来处理请求和响应的数据。

```
app.get('/', function (req, res) {
   // --
})
```

request 和 **response** 对象的具体介绍：

Request 对象 - request 对象表示 HTTP 请求，包含了请求查询字符串，参数，内容，HTTP 头部等属性。常见属性有：

1. req.app：当callback为外部文件时，用req.app访问express的实例
2. req.baseUrl：获取路由当前安装的URL路径
3. req.body / req.cookies：获得「请求主体」/ Cookies
4. req.fresh / req.stale：判断请求是否还「新鲜」
5. req.hostname / req.ip：获取主机名和IP地址
6. req.originalUrl：获取原始请求URL
7. req.params：获取路由的parameters
8. req.path：获取请求路径
9. req.protocol：获取协议类型
10. req.query：获取URL的查询参数串
11. req.route：获取当前匹配的路由
12. req.subdomains：获取子域名
13. req.accpets（）：检查请求的Accept头的请求类型
14. req.acceptsCharsets / req.acceptsEncodings / req.acceptsLanguages
15. req.get（）：获取指定的HTTP请求头
16. req.is（）：判断请求头Content-Type的MIME类型

Response 对象 - response 对象表示 HTTP 响应，即在接收到请求时向客户端发送的 HTTP 响应数据。常见属性有：

1. res.app：同req.app一样
2. res.append（）：追加指定HTTP头
3. res.set（）在res.append（）后将重置之前设置的头
4. res.cookie（name，value [，option]）：设置Cookie
5. opition: domain / expires / httpOnly / maxAge / path / secure / signed
6. res.clearCookie（）：清除Cookie
7. res.download（）：传送指定路径的文件
8. res.get（）：返回指定的HTTP头
9. res.json（）：传送JSON响应
10. res.jsonp（）：传送JSONP响应
11. res.location（）：只设置响应的Location HTTP头，不设置状态码或者close response
12. res.redirect（）：设置响应的Location HTTP头，并且设置状态码302
13. res.send（）：传送HTTP响应
14. res.sendFile（path [，options] [，fn]）：传送指定路径的文件 -会自动根据文件

extension设定Content-Type

15. res.set（）：设置HTTP头，传入object可以一次设置多个头
16. res.status（）：设置HTTP状态码
17. res.type（）：设置Content-Type的MIME类型

路由

我们已经了解了 HTTP 请求的基本应用，而路由决定了由谁(指定脚本)去响应客户端请求。

在HTTP请求中，我们可以通过路由提取出请求的URL以及GET/POST参数。

接下来我们扩展 Hello World，添加一些功能来处理更多类型的 HTTP 请求。

创建 express_demo2.js 文件，代码如下所示：

```
var express = require('express');
var app = express();

//  主页输出 "Hello World"
app.get('/', function (req, res) {
   console.log("主页 GET 请求");
   res.send('Hello GET');
})

//  POST 请求
app.post('/', function (req, res) {
   console.log("主页 POST 请求");
   res.send('Hello POST');
})

//  /del_user 页面响应
app.delete('/del_user', function (req, res) {
   console.log("/del_user 响应 DELETE 请求");
   res.send('删除页面');
})

//  /list_user 页面 GET 请求
app.get('/list_user', function (req, res) {
   console.log("/list_user GET 请求");
   res.send('用户列表页面');
})

// 对页面 abcd, abxcd, ab123cd, 等响应 GET 请求
app.get('/ab*cd', function(req, res) {
   console.log("/ab*cd GET 请求");
   res.send('正则匹配');
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

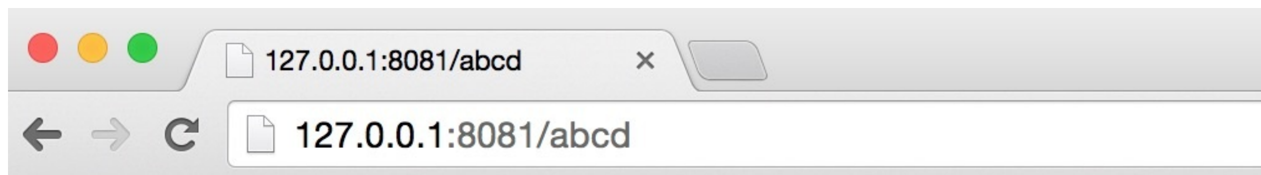
})
```

执行以上代码：

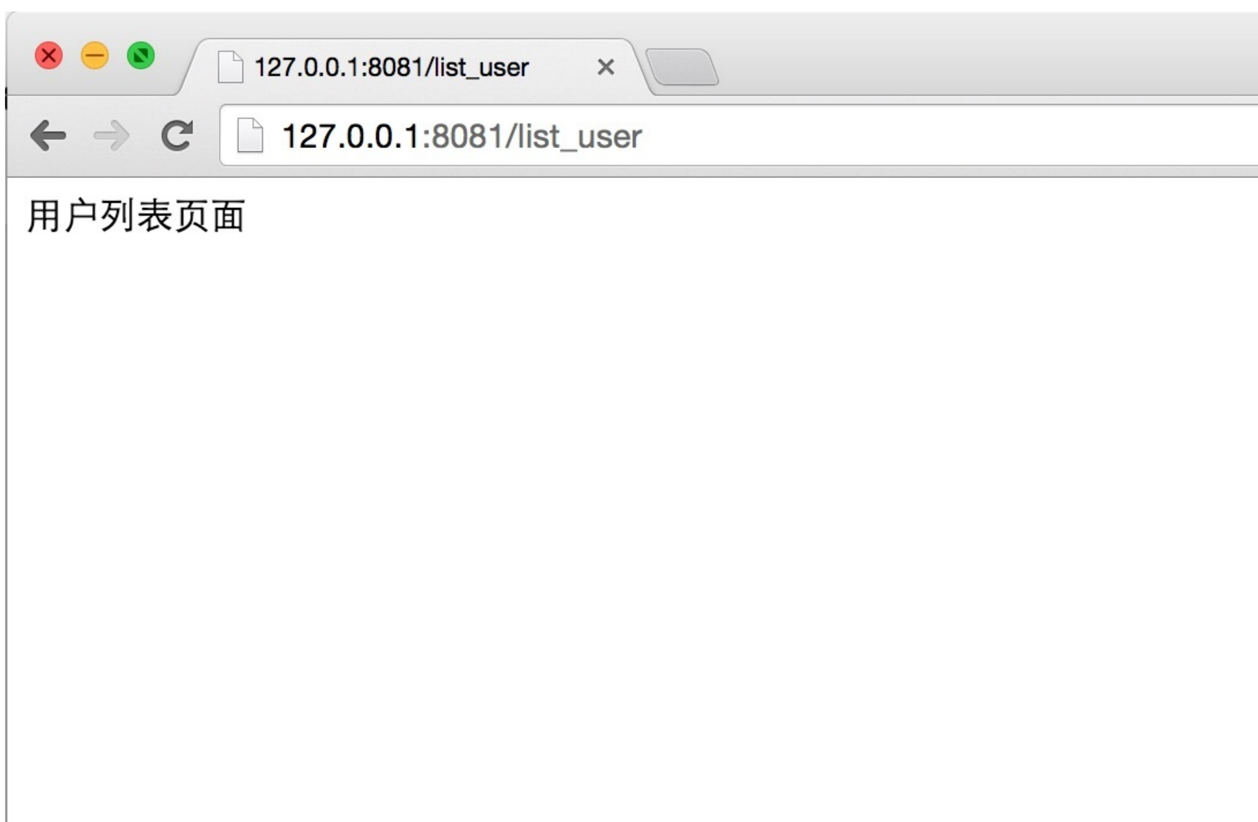
```
$ node express_demo2.js
应用实例，访问地址为 http://0.0.0.0:8081
```

接下来你可以尝试访问 http://127.0.0.1:8081 不同的地址，查看效果。

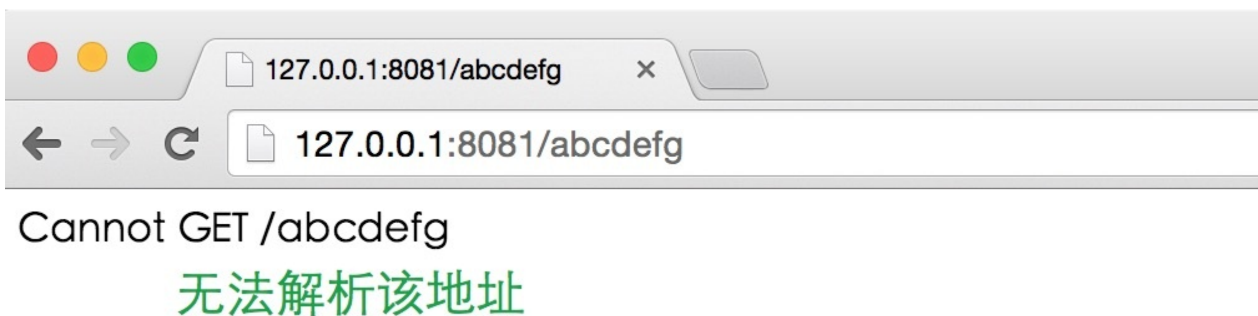
在浏览器中访问 http://127.0.0.1:8081/list_user，结果如下图所示：



在浏览器中访问 http://127.0.0.1:8081/abcd，结果如下图所示：



在浏览器中访问 http://127.0.0.1:8081/abcdefg，结果如下图所示：



静态文件

Express 提供了内置的中间件 **express.static** 来设置静态文件如：图片， CSS, JavaScript 等。

你可以使用 **express.static** 中间件来设置静态文件路径。例如，如果你将图片， CSS, JavaScript 文件放在 public 目录下，你可以这么写：

```
app.use(express.static('public'));
```

我们可以到 public/images 目录下放些图片,如下所示：

```
node_modules
server.js
public/
public/images
public/images/logo.png
```

让我们再修改下 "Hello Word" 应用添加处理静态文件的功能。

创建 express_demo3.js 文件，代码如下所示：

```
var express = require('express');
var app = express();

app.use(express.static('public'));

app.get('/', function (req, res) {
   res.send('Hello World');
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

执行以上代码：

```
$ node express_demo3.js
应用实例，访问地址为 http://0.0.0.0:8081
```

执行以上代码：

在浏览器中访问 http://127.0.0.1:8081/images/logo.png（本实例采用了菜鸟教程的logo），结果如下图所示：

RUNOOB.COM

GET 方法

以下实例演示了在表单中通过 GET 方法提交两个参数，我们可以使用 server.js 文件内的 **process_get** 路由器来处理输入：

index.htm 文件代码如下：

```
<html>
<body>
<form action="http://127.0.0.1:8081/process_get" method="GET">
First Name: <input type="text" name="first_name">  <br>

Last Name: <input type="text" name="last_name">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

server.js 文件代码如下:

```
var express = require('express');
var app = express();

app.use(express.static('public'));

app.get('/index.htm', function (req, res) {
   res.sendFile( __dirname + "/" + "index.htm" );
})

app.get('/process_get', function (req, res) {

   // 输出 JSON 格式
   response = {
       first_name:req.query.first_name,
       last_name:req.query.last_name
   };
   console.log(response);
   res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

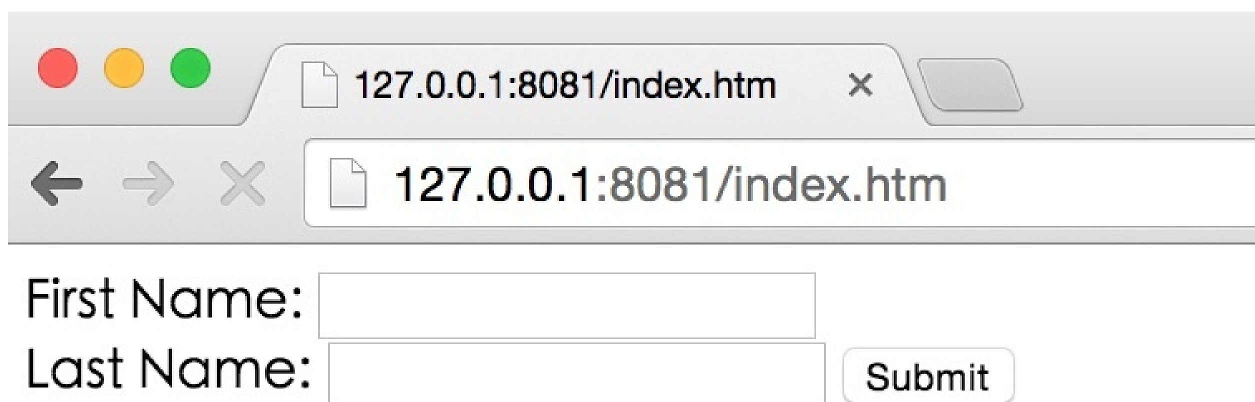
  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

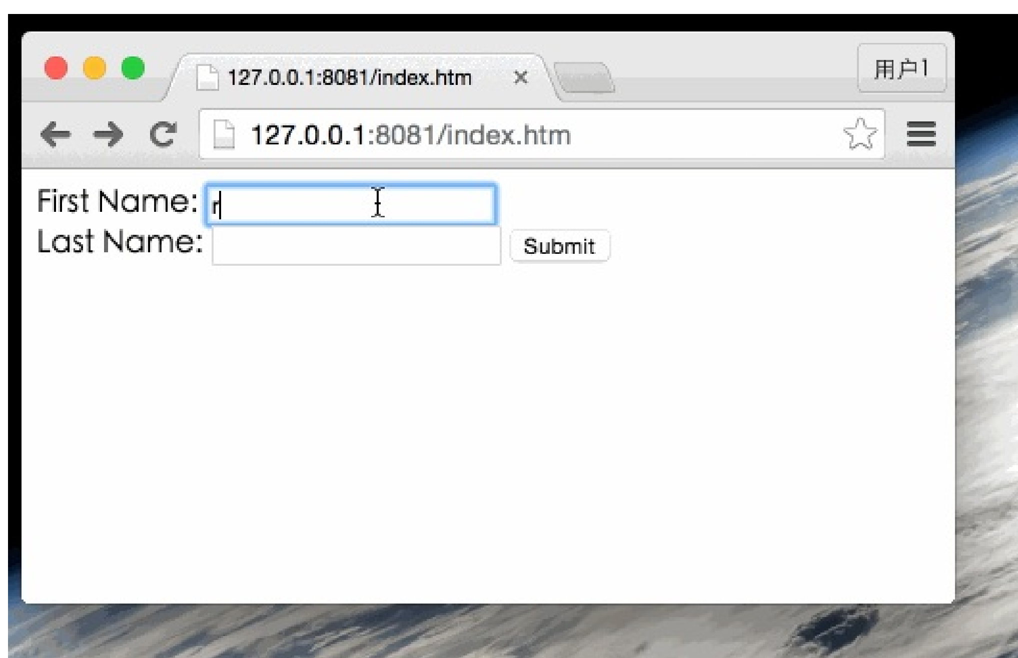
执行以上代码：

```
node server.js
应用实例，访问地址为 http://0.0.0.0:8081
```

浏览器访问 http://127.0.0.1:8081/index.htm，如图所示：



现在你可以向表单输入数据，并提交，如下演示：



POST 方法

以下实例演示了在表单中通过 POST 方法提交两个参数，我们可以使用 server.js 文件内的 **process_get** 路由器来处理输入：

index.htm 文件代码修改如下：

```
<html>
<body>
<form action="http://127.0.0.1:8081/process_post" method="POST">
First Name: <input type="text" name="first_name">  <br>

Last Name: <input type="text" name="last_name">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

server.js 文件代码修改如下:

```
var express = require('express');
var app = express();
var bodyParser = require('body-parser');

// 创建 application/x-www-form-urlencoded 编码解析
var urlencodedParser = bodyParser.urlencoded({ extended: false })

app.use(express.static('public'));

app.get('/index.htm', function (req, res) {
   res.sendFile( __dirname + "/" + "index.htm" );
})

app.post('/process_post', urlencodedParser, function (req, res) {

   // 输出 JSON 格式
   response = {
       first_name:req.body.first_name,
       last_name:req.body.last_name
   };
   console.log(response);
   res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

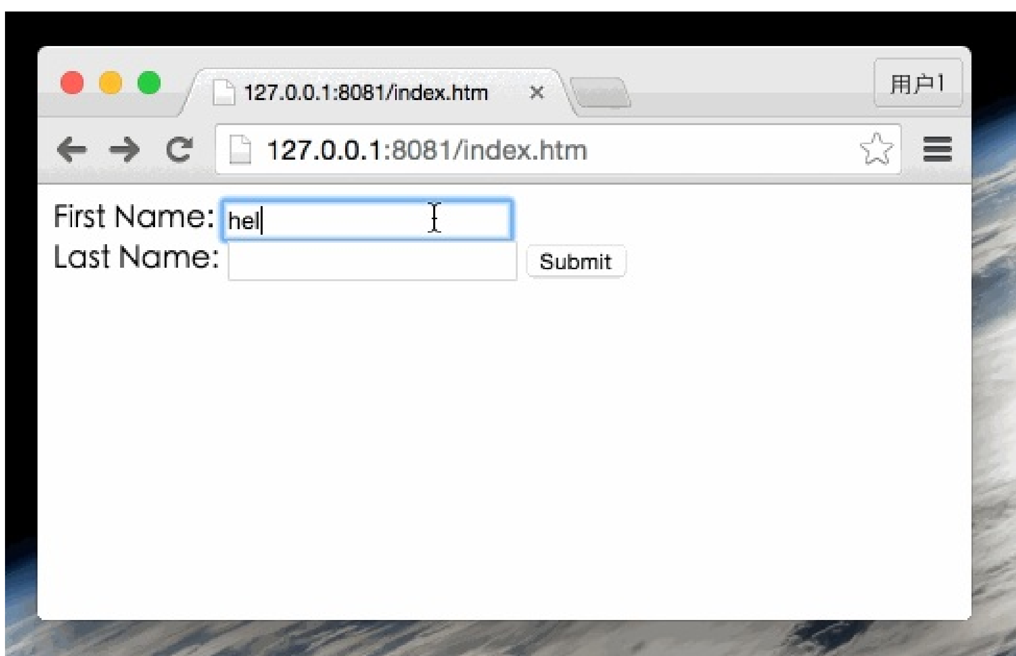
})
```

执行以上代码：

```
$ node express_demo.js
应用实例，访问地址为 http://0.0.0.0:8081
```

浏览器访问 http://127.0.0.1:8081/index.htm，如图所示：

现在你可以向表单输入数据，并提交，如下演示：



文件上传

以下我们创建一个用于上传文件的表单，使用 POST 方法，表单 enctype 属性设置为 multipart/form-data。

index.htm 文件代码修改如下：

```
<html>
<head>
<title>文件上传表单</title>
</head>
<body>
<h3>文件上传：</h3>
选择一个文件上传: <br />
<form action="/file_upload" method="post" enctype="multipart/form-data">
<input type="file" name="image" size="50" />
<br />
<input type="submit" value="上传文件" />
</form>
</body>
</html>
```

server.js 文件代码修改如下:

```
var express = require('express');
var app = express();
var fs = require("fs");

var bodyParser = require('body-parser');
var multer  = require('multer');

app.use(express.static('public'));
app.use(bodyParser.urlencoded({ extended: false }));
app.use(multer({ dest: '/tmp/'}).array('image'));

app.get('/index.htm', function (req, res) {
   res.sendFile( __dirname + "/" + "index.htm" );
})

app.post('/file_upload', function (req, res) {

   console.log(req.files[0]);  // 上传的文件信息

   var des_file = __dirname + "/" + req.files[0].originalname;
   fs.readFile( req.files[0].path, function (err, data) {
        fs.writeFile(des_file, data, function (err) {
         if( err ){
              console.log( err );
         }else{
               response = {
                   message:'File uploaded successfully',
                   filename:req.files[0].originalname
              };
          }
          console.log( response );
          res.end( JSON.stringify( response ) );
       });
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

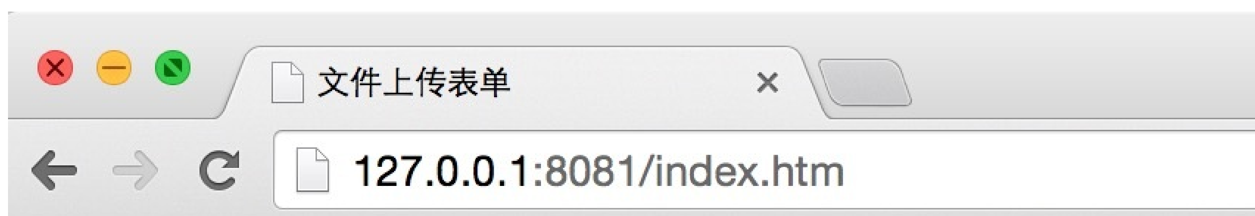
  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

执行以上代码：

```
$ node express_demo.js
应用实例，访问地址为 http://0.0.0.0:8081
```

浏览器访问 http://127.0.0.1:8081/index.htm，如图所示：



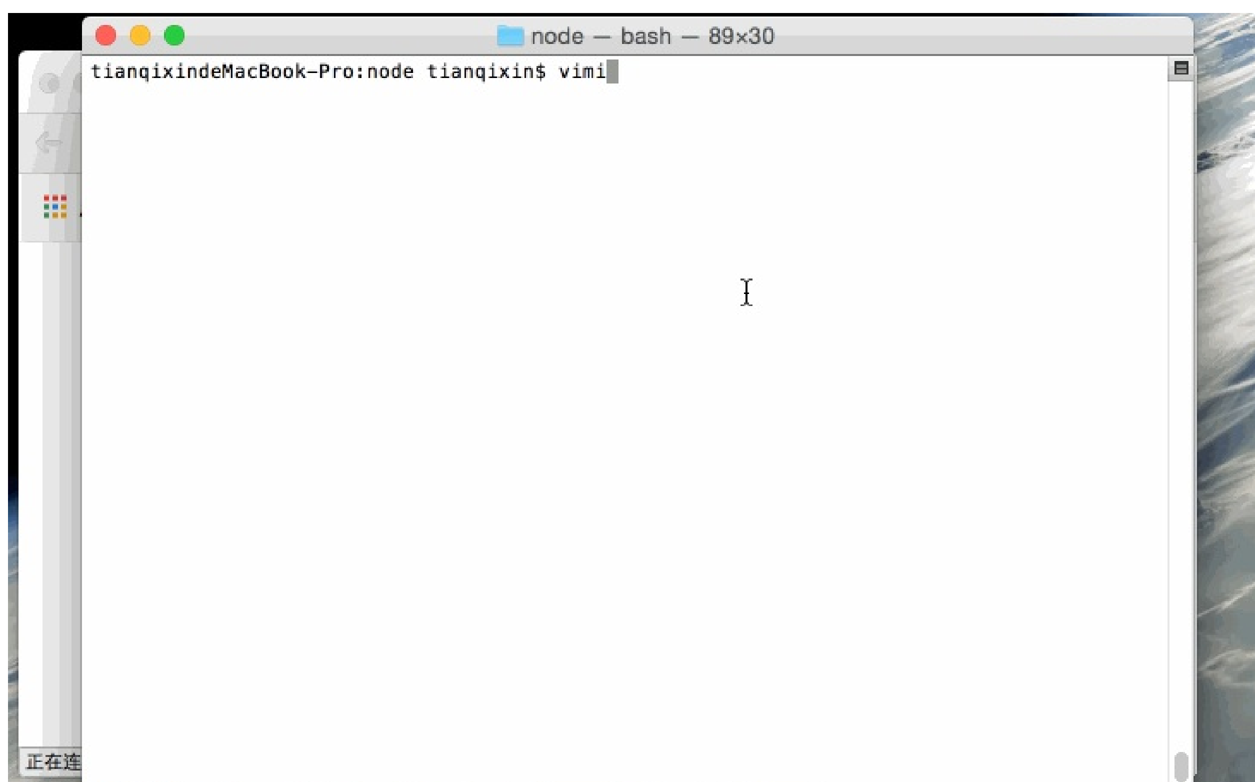
文件上传：

选择一个文件上传:

选择文件 未选择任何文件

上传文件

现在你可以向表单输入数据，并提交，如下演示：



Cookie 管理

我们可以使用中间件向 Node.js 服务器发送 cookie 信息，以下代码输出了客户端发送的 cookie 信息：

```
// express_cookie.js 文件
var express      = require('express')
var cookieParser = require('cookie-parser')

var app = express()
app.use(cookieParser())

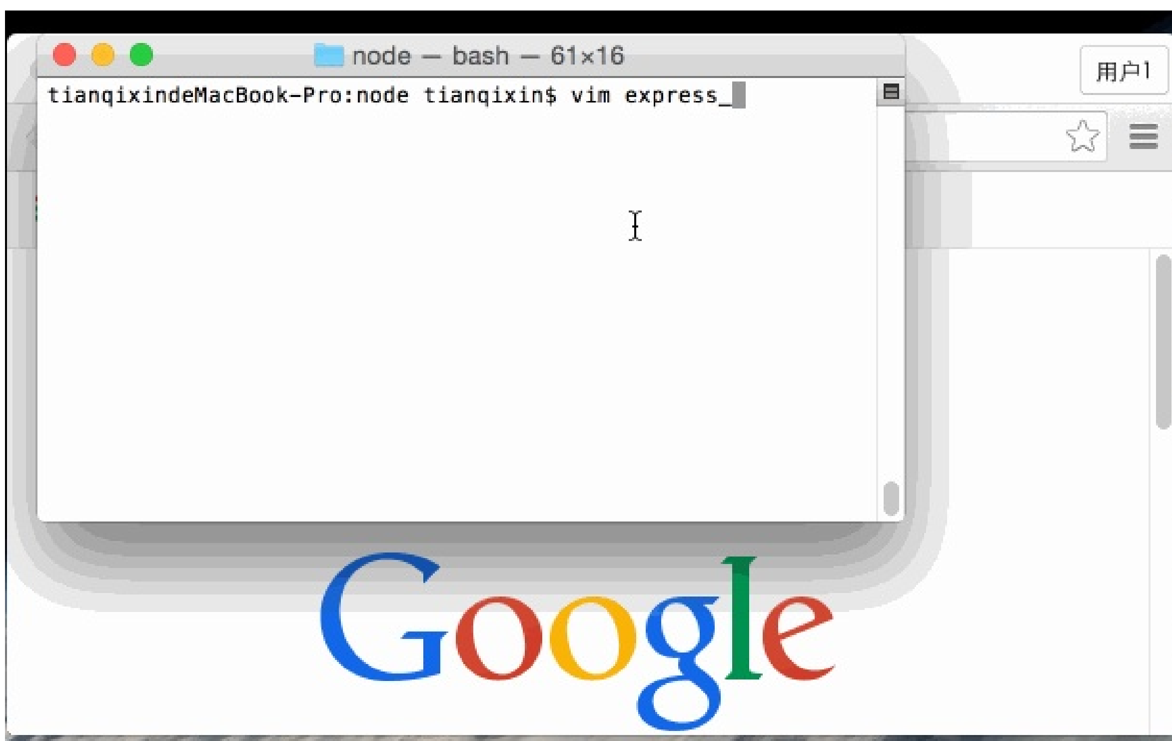
app.get('/', function(req, res) {
  console.log("Cookies: ", req.cookies)
})

app.listen(8081)
```

执行以上代码：

```
$ node express_demo.js
```

现在你可以访问 http://127.0.0.1:8081 并查看终端信息的输出，如下演示：



Node.js RESTful API

什么是 REST？

RREST即表述性状态传递（英文：Representational State Transfer，简称REST）是Roy Fielding博士在2000年他的博士论文中提出来的一种软件架构风格。

表述性状态转移是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是RESTful。需要注意的是，REST是设计风格而不是标准。REST通常基于使用HTTP，URI，和XML（标准通用标记语言下的一个子集）以及HTML（标准通用标记语言下的一个应用）这些现有的广泛流行的协议和标准。REST 通常使用 JSON 数据格式。

HTTP 方法

以下为 REST 基本架构的四个方法：

- **GET** - 用于获取数据。
- **PUT** - 用于添加数据。
- **DELETE** - 用于删除数据。
- **POST** - 用于更新或添加数据。

RESTful Web Services

Web service是一个平台独立的，低耦合的，自包含的、基于可编程的web的应用程序，可使用开放的XML（标准通用标记语言下的一个子集）标准来描述、发布、发现、协调和配置这些应用程序，用于开发分布式的互操作的应用程序。

基于 REST 架构的 Web Services 即是 RESTful。

由于轻量级以及通过 HTTP 直接传输数据的特性，Web 服务的 RESTful 方法已经成为最常见的替代方法。可以使用各种语言（比如 Java 程序、Perl、Ruby、Python、PHP 和 Javascript[包括 Ajax]）实现客户端。

RESTful Web 服务通常可以通过自动客户端或代表用户的应用程序访问。但是，这种服务的简便性让用户能够与之直接交互，使用它们的 Web 浏览器构建一个 GET URL 并读取返回的内容。

更多介绍，可以查看：RESTful 架构详解

创建 RESTful

首先，创建一个 json 数据资源文件 users.json，内容如下：

```
{
   "user1" : {
      "name" : "mahesh",
      "password" : "password1",
      "profession" : "teacher",
      "id": 1
   },
   "user2" : {
      "name" : "suresh",
      "password" : "password2",
      "profession" : "librarian",
      "id": 2
   },
   "user3" : {
      "name" : "ramesh",
      "password" : "password3",
      "profession" : "clerk",
      "id": 3
   }
}
```

基于以上数据，我们创建以下 RESTful API：

序号	URI	HTTP 方法	发送内容	结果
1	listUsers	GET	空	显示所有用户列表
2	addUser	POST	JSON 字符串	添加新用户
3	deleteUser	DELETE	JSON 字符串	删除用户
4	:id	GET	空	显示用户详细信息

获取用户列表：

以下代码，我们创建了 RESTful API **listUsers**，用于读取用户的信息列表， server.js 文件代码如下所示：

```
var express = require('express');
var app = express();
var fs = require("fs");

app.get('/listUsers', function (req, res) {
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       console.log( data );
       res.end( data );
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

接下来执行以下命令：

```
$ node server.js
应用实例，访问地址为 http://0.0.0.0:8081
```

在浏览器中访问 http://127.0.0.1:8081/listUsers，结果如下所示：

```
{
   "user1" : {
      "name" : "mahesh",
      "password" : "password1",
      "profession" : "teacher",
      "id": 1
   },
   "user2" : {
      "name" : "suresh",
      "password" : "password2",
      "profession" : "librarian",
      "id": 2
   },
   "user3" : {
      "name" : "ramesh",
      "password" : "password3",
      "profession" : "clerk",
      "id": 3
   }
}
```

添加用户

以下代码，我们创建了 RESTful API **addUser**， 用于添加新的用户数据，server.js 文件代码如下所示：

```
var express = require('express');
var app = express();
var fs = require("fs");

//添加的新用户数据
var user = {
   "user4" : {
      "name" : "mohit",
      "password" : "password4",
      "profession" : "teacher",
      "id": 4
   }
}

app.get('/addUser', function (req, res) {
   // 读取已存在的数据
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       data = JSON.parse( data );
       data["user4"] = user["user4"];
       console.log( data );
       res.end( JSON.stringify(data));
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port
  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

接下来执行以下命令：

```
$ node server.js
应用实例，访问地址为 http://0.0.0.0:8081
```

在浏览器中访问 http://127.0.0.1:8081/addUsers，结果如下所示：

```
{ user1:
   { name: 'mahesh',
     password: 'password1',
     profession: 'teacher',
     id: 1 },
  user2:
   { name: 'suresh',
     password: 'password2',
     profession: 'librarian',
     id: 2 },
  user3:
   { name: 'ramesh',
     password: 'password3',
     profession: 'clerk',
     id: 3 },
  user4:
   { name: 'mohit',
     password: 'password4',
     profession: 'teacher',
     id: 4 }
}
```

显示用户详情

以下代码，我们创建了 RESTful API **:id**（用户**id**）， 用于读取指定用户的详细信息，server.js 文件代码如下所示：

```
var express = require('express');
var app = express();
var fs = require("fs");

app.get('/:id', function (req, res) {
   // 首先我们读取已存在的用户
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       data = JSON.parse( data );
       var user = data["user" + req.params.id]
       console.log( user );
       res.end( JSON.stringify(user));
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port
  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

接下来执行以下命令：

```
$ node server.js
应用实例，访问地址为 http://0.0.0.0:8081
```

在浏览器中访问 http://127.0.0.1:8081/2，结果如下所示：

```
{
   "name":"suresh",
   "password":"password2",
   "profession":"librarian",
   "id":2
}
```

删除用户

以下代码，我们创建了 RESTful API **deleteUser**， 用于删除指定用户的详细信息，以下实例中，用户 id 为 2，server.js 文件代码如下所示：

```
var express = require('express');
var app = express();
var fs = require("fs");

var id = 2;

app.get('/deleteUser', function (req, res) {

   // First read existing users.
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       data = JSON.parse( data );
       delete data["user" + 2];

       console.log( data );
       res.end( JSON.stringify(data));
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port
  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

接下来执行以下命令：

```
$ node server.js
应用实例，访问地址为 http://0.0.0.0:8081
```

在浏览器中访问 http://127.0.0.1:8081/deleteUser，结果如下所示：

```
{ user1:
   { name: 'mahesh',
     password: 'password1',
     profession: 'teacher',
     id: 1 },
  user3:
   { name: 'ramesh',
     password: 'password3',
     profession: 'clerk',
     id: 3 }
}
```

Node.js 多进程

我们都知道 Node.js 是以单线程的模式运行的，但它使用的是事件驱动来处理并发，这样有助于我们在多核 cpu 的系统上创建多个子进程，从而提高性能。

每个子进程总是带有三个流对象：child.stdin, child.stdout 和child.stderr。他们可能会共享父进程的 stdio 流，或者也可以是独立的被导流的流对象。

Node 提供了 child_process 模块来创建子进程，方法有：

- **exec** - child_process.exec 使用子进程执行命令，缓存子进程的输出，并将子进程的输出以回调函数参数的形式返回。
- **spawn** - child_process.spawn 使用指定的命令行参数创建新线程。
- **fork** - child_process.fork 是 spawn()的特殊形式，用于在子进程中运行的模块，如 fork('./son.js') 相当于 spawn('node', ['./son.js']) 。与spawn方法不同的是，fork会在父进程与子进程之间，建立一个通信管道，用于进程之间的通信。

exec() 方法

child_process.exec 使用子进程执行命令，缓存子进程的输出，并将子进程的输出以回调函数参数的形式返回。

语法如下所示：

```
child_process.exec(command[, options], callback)
```

参数

参数说明如下：

command： 字符串， 将要运行的命令，参数使用空格隔开

options ：对象，可以是：

- cwd，字符串，子进程的当前工作目录
- env，对象 环境变量键值对
- encoding，字符串，字符编码（默认：'utf8'）
- shell，字符串，将要执行命令的 Shell（默认: 在 UNIX 中为 `/bin/sh`， 在 Windows 中为 `cmd.exe`， Shell 应当能识别 `-c` 开关在 UNIX 中，或 `/s /c` 在 Windows 中。 在 Windows 中，命令行解析应当能兼容 `cmd.exe`）

- timeout，数字，超时时间（默认： 0）
- maxBuffer，数字， 在 stdout 或 stderr 中允许存在的最大缓冲（二进制），如果超出那么子进程将会被杀死 （默认: 200*1024）
- killSignal ，字符串，结束信号（默认：'SIGTERM'）
- uid，数字，设置用户进程的 ID
- gid，数字，设置进程组的 ID

callback ：回调函数，包含三个参数error, stdout 和 stderr。

exec() 方法返回最大的缓冲区，并等待进程结束，一次性返回缓冲区的内容。

实例

让我们创建两个 js 文件 support.js 和 master.js。

support.js 文件代码：

```
console.log("进程 " + process.argv[2] + " 执行。" );
```

master.js 文件代码：

```
const fs = require('fs');
const child_process = require('child_process');

for(var i=0; i<3; i++) {
   var workerProcess = child_process.exec('node support.js '+i,
      function (error, stdout, stderr) {
         if (error) {
            console.log(error.stack);
            console.log('Error code: '+error.code);
            console.log('Signal received: '+error.signal);
         }
         console.log('stdout: ' + stdout);
         console.log('stderr: ' + stderr);
      });

      workerProcess.on('exit', function (code) {
      console.log('子进程已退出，退出码 '+code);
   });
}
```

执行以上代码，输出结果为：

```
$ node master.js
子进程已退出，退出码 0
stdout: 进程 1 执行。

stderr:
子进程已退出，退出码 0
stdout: 进程 0 执行。

stderr:
子进程已退出，退出码 0
stdout: 进程 2 执行。

stderr:
```

spawn() 方法

child_process.spawn 使用指定的命令行参数创建新线程，语法格式如下：

```
child_process.spawn(command[, args][, options])
```

参数

参数说明如下：

command： 将要运行的命令

args： Array 字符串参数数组

options Object

- cwd String 子进程的当前工作目录
- env Object 环境变量键值对
- stdio Array|String 子进程的 stdio 配置
- detached Boolean 这个子进程将会变成进程组的领导
- uid Number 设置用户进程的 ID
- gid Number 设置进程组的 ID

spawn() 方法返回流 (stdout & stderr)，在进程返回大量数据时使用。进程一旦开始执行时 spawn() 就开始接收响应。

实例

让我们创建两个 js 文件 support.js 和 master.js。

support.js 文件代码：

```
console.log("进程 " + process.argv[2] + " 执行。" );
```

master.js 文件代码：

```
const fs = require('fs');
const child_process = require('child_process');

for(var i=0; i<3; i++) {
   var workerProcess = child_process.spawn('node', ['support.js', i]);

   workerProcess.stdout.on('data', function (data) {
      console.log('stdout: ' + data);
   });

   workerProcess.stderr.on('data', function (data) {
      console.log('stderr: ' + data);
   });

   workerProcess.on('close', function (code) {
      console.log('子进程已退出，退出码 '+code);
   });
}
```

执行以上代码，输出结果为：

```
$ node master.js stdout: 进程 0 执行。

子进程已退出，退出码 0
stdout: 进程 1 执行。

子进程已退出，退出码 0
stdout: 进程 2 执行。

子进程已退出，退出码 0
```

fork 方法

child_process.fork 是 spawn() 方法的特殊形式，用于创建进程，语法格式如下：

```
child_process.fork(modulePath[, args][, options])
```

参数

参数说明如下：

modulePath： String，将要在子进程中运行的模块

args： Array 字符串参数数组

options：Object

- cwd String 子进程的当前工作目录
- env Object 环境变量键值对
- execPath String 创建子进程的可执行文件
- execArgv Array 子进程的可执行文件的字符串参数数组（默认： process.execArgv）

- silent Boolean 如果为 `true`，子进程的 `stdin`，`stdout` 和 `stderr` 将会被关联至父进程，否则，它们将会从父进程中继承。（默认为：`false`）
- uid Number 设置用户进程的 ID
- gid Number 设置进程组的 ID

返回的对象除了拥有ChildProcess实例的所有方法，还有一个内建的通信信道。

h3>实例

让我们创建两个 js 文件 support.js 和 master.js。

support.js 文件代码：

```
console.log("进程 " + process.argv[2] + " 执行。" );
```

master.js 文件代码：

```
const fs = require('fs');
const child_process = require('child_process');

for(var i=0; i<3; i++) {
   var worker_process = child_process.fork("support.js", [i]);

   worker_process.on('close', function (code) {
      console.log('子进程已退出，退出码 ' + code);
   });
}
```

执行以上代码，输出结果为：

```
$ node master.js
进程 0 执行。
子进程已退出，退出码 0
进程 1 执行。
子进程已退出，退出码 0
进程 2 执行。
子进程已退出，退出码 0
```

Node.js JXcore 打包

Node.js 是一个开放源代码、跨平台的、用于服务器端和网络应用的运行环境。

JXcore 是一个支持多线程的 Node.js 发行版本，基本不需要对你现有的代码做任何改动就可以直接线程安全地以多线程运行。

但我们这篇文章主要是要教大家介绍 JXcore 的打包功能。

JXcore 安装

下载 JXcore 安装包，并解压，在解压的的目录下提供了 jx 二进制文件命令，接下来我们主要使用这个命令。

步骤1、下载

下载 JXcore 安装包 http://jxcore.com/downloads/，你需要根据你自己的系统环境来下载安装包。

1、Window 平台下载：Download，

2、Linux/OSX 下载安装命令，直接下载解压包下的 jx 二进制文件拷贝到 /usr/bin 目录下：

```
$ wget https://s3.amazonaws.com/nodejx/jx_rh64.zip
$ unzip jx_rh64.zip
$ cp jx_rh64/jx /usr/bin
```

将 /usr/bin 添加到 PATH 路径中：

```
$ export PATH=$PATH:/usr/bin
```

以上步骤如果操作正确，使用以下命令，会输出版本号信息：

```
$ jx --version
v0.10.32
```

包代码

例如，我们的 Node.js 项目包含以下几个文件，其中 index.js 是主文件：

```
drwxr-xr-x  2 root root  4096 Nov 13 12:42 images
-rwxr-xr-x  1 root root 30457 Mar  6 12:19 index.htm
-rwxr-xr-x  1 root root 30452 Mar  1 12:54 index.js
drwxr-xr-x 23 root root  4096 Jan 15 03:48 node_modules
drwxr-xr-x  2 root root  4096 Mar 21 06:10 scripts
drwxr-xr-x  2 root root  4096 Feb 15 11:56 style
```

接下来我们使用 **jx** 命令打包以上项目，并指定 index.js 为 Node.js 项目的主文件：

```
$ jx package index.js index
```

以上命令执行成功，会生成以下两个文件：

- **index.jxp** 这是一个中间件文件，包含了需要编译的完整项目信息。
- **index.jx** 这是一个完整包信息的二进制文件，可运行在客户端上。

载入 JX 文件

我们使用 **jx** 命令打包项目：

```
$ node index.js command_line_arguments
```

使用 JXcore 编译后，我们可以使用以下命令来执行生成的 jx 二进制文件：

```
$ jx index.jx command_line_arguments
```

更多 JXcore 功能特性你可以参考官网：http://jxcore.com/。

免责声明

W3School提供的内容仅用于培训。我们不保证内容的正确性。通过使用本站内容随之而来的风险与本站无关。W3School简体中文版的所有内容仅供测试，对任何法律问题及风险不承担任何责任。